



INSIDE MACINTOSH

QuickDraw GX Objects



Addison-Wesley Publishing Company

Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam Bonn
Sydney Singapore Tokyo Madrid San Juan
Paris Seoul Milan Mexico City Taipei

Apple Computer, Inc.
© 1994 Apple Computer, Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Computer, Inc. Printed in the United States of America.

No licenses, express or implied, are granted with respect to any of the technology described in this book. Apple retains all intellectual property rights associated with the technology described in this book. This book is intended to assist application developers to develop applications only for Apple Macintosh computers.

Every effort has been made to ensure that the information in this manual is accurate. Apple is not responsible for printing or clerical errors.

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, LaserWriter, and Macintosh are trademarks of Apple Computer, Inc., registered in the United States and other countries.

Adobe Illustrator, Adobe Photoshop, and PostScript are trademarks of Adobe Systems Incorporated, which may be registered in certain jurisdictions.

America Online is a service mark of Quantum Computer Services, Inc.

CompuServe is a registered service mark of CompuServe, Inc.

FrameMaker is a registered trademark of Frame Technology Corporation.

Helvetica, Times, and Palatino are registered trademarks of Linotype Company.

ITC Zapf Dingbats is a registered trademark of International Typeface Corporation.

Optrotech is a trademark of Orbotech Corporation.

Pantone is a registered trademark of Pantone, Inc.

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

ISBN 0-201-40675-6
1 2 3 4 5 6 7 8 9-CRW-9897969594
First Printing, April 1994

Library of Congress Cataloging-in-Publication Data

Inside Macintosh : QuickDraw GX object / [Apple Computer, Inc.].

p. cm.

Includes index.

ISBN 0-201-40675-6

1. Macintosh (Computer) 2. Computer graphics. 3. QuickDraw GX.

I. Apple Computer, Inc.

QA76.8.M3156228 1994

006.6'765—dc20

94-1843
CIP

Contents

	Figures, Tables, and Listings	xiii
Preface	About This Book	xix
	What to Read	xx
	Chapter Organization	xxi
	Conventions Used in This Book	xxii
	Special Fonts	xxii
	Types of Notes	xxii
	Numerical Formats	xxii
	Type Definitions for Enumerations	xxiii
	Illustrations	xxiii
	Development Environment	xxiii
	Developer Products and Support	xxiv
Chapter 1	Introduction to QuickDraw GX	1-1
	What Is QuickDraw GX?	1-3
	Color Graphics	1-4
	Typography	1-5
	Printing	1-6
	What QuickDraw GX Is Not	1-7
	QuickDraw GX Objects	1-7
	How QuickDraw GX Defines Objects	1-8
	Advantages of an Object-Based Structure	1-9
	Kinds of QuickDraw GX Objects	1-10
	Shape Objects	1-10
	Supporting Objects	1-11
	Printing Objects	1-14
	Object Properties	1-15
	Default Objects and Default Properties	1-17
	Adding Custom Behavior With Tag Objects	1-17
	Objects and Memory	1-18
	Application Memory and QuickDraw GX Memory	1-18
	Sharing and Multiple Object References	1-19
	Owner Count	1-20
	Cloning	1-20
	Automatic Loading and Unloading of Objects	1-21
	Direct Access to Object Structure: Locking and Unlocking	1-22
	External Storage of Objects: Flattening and Unflattening	1-23

Drawing and Hit-Testing Shapes	1-23
Drawing	1-24
Mapping and Clipping	1-24
View-Related Objects	1-25
The Drawing Sequence: Coordinate Conversion	1-28
Hit-Testing	1-32
Printing With QuickDraw GX	1-34
Core Printing Features	1-35
Custom Dialog Boxes and Page Formats	1-36
Advanced Printing Features	1-37
The QuickDraw GX Programming Environment	1-38
Setting Up QuickDraw GX Memory	1-38
Handling Errors	1-38
Debugging	1-39
Debugging and Non-Debugging Versions	1-39
Debugging With GraphicsBug	1-40
Programming Conventions and Consistencies	1-41
Object Behavior	1-41
Functions and Function Results	1-41
Function Parameters	1-42
Code Naming Conventions	1-44
Relationship to the Macintosh Toolbox	1-44
Summary Table and Diagram of QuickDraw GX Objects	1-45

Chapter 2

Shape Objects 2-1

About QuickDraw GX Shapes	2-5
About Shape Objects	2-7
Shape Properties	2-7
Shape Type	2-9
Shape Geometry	2-11
Shape Fill	2-13
Shape Attributes	2-16
Default Shapes	2-18
Modifying Shape Properties	2-19
Drawing Shapes	2-20
Hit-Testing Shapes	2-20
Saving and Restoring Shapes	2-22
Using Shape Objects	2-22
Creating and Manipulating Shape Objects	2-22
Getting and Setting the Default Shape Objects	2-23
Creating and Disposing of Shape Objects	2-24
Getting the Size of a Shape Object in Memory	2-25
Copying, Comparing, and Cloning Shape Objects	2-25
Caching Shape Objects	2-27
Loading and Unloading Shape Objects	2-27

Manipulating Shape Object Properties	2-28	
Getting and Setting a Shape Object's Type, Fill, and Attributes		2-28
Copying the Geometry From One Shape to Another	2-29	
Getting and Setting a Shape Object's Style, Ink, and Transform		2-30
Resetting a Shape Object's Properties to Their Default Values		2-31
Manipulating a Shape Object's Owner Count	2-31	
Getting and Setting a Shape Object's Tag References		2-32
Converting Shapes From One Type to Another	2-32	
Directly Manipulating a Shape's Geometry	2-34	
Drawing and Hit-Testing Shapes	2-35	
Drawing Shapes	2-35	
Hit-Testing Shapes	2-36	
Flattening and Unflattening Shapes	2-39	
Shape-Related Functions Described Elsewhere	2-42	
Shape Objects Reference	2-45	
Constants and Data Types	2-45	
The Shape Object	2-46	
Shape Type	2-46	
Shape Fill	2-46	
Shape Attributes	2-47	
Flatten Flags	2-48	
The Spool Block	2-49	
The Hit-Test Info Structure	2-50	
Functions	2-51	
Creating and Manipulating Shape Objects	2-52	
Manipulating Shape Object Properties	2-65	
Directly Manipulating a Shape's Geometry	2-80	
Drawing and Hit-Testing Shapes	2-84	
Flattening and Unflattening Shape Objects	2-87	
Application-Defined Spool Function	2-91	
Summary of Shape Objects	2-93	
Constants and Data Types	2-93	
Functions	2-95	
Application-Defined Spool Function	2-97	

Chapter 3

Style Objects 3-1

About Style Objects	3-3	
Style Object Properties	3-4	
The Default Style Object	3-6	
Using Style Objects	3-7	
Creating and Manipulating Style Objects	3-7	
Creating and Deleting a Style Object	3-7	
Copying, Comparing, and Cloning Style Objects	3-8	
Loading and Unloading Style Objects	3-10	

Manipulating Style Object Properties	3-10
Resetting a Style Object's Default Properties	3-11
Getting and Setting Style Attributes and Text Attributes	3-11
Manipulating a Style Object's Owner Count	3-11
Getting and Setting a Style Object's Tag References	3-14
Style-Related Functions Described Elsewhere	3-14
Style Objects Reference	3-15
Constants and Data Types	3-16
The Style Object	3-16
Functions	3-16
Creating and Manipulating Style Objects	3-16
Manipulating Style Object Properties	3-21
Summary of Style Objects	3-26
Constants and Data Types	3-26
Functions	3-26

Chapter 4

Colors and Color-Related Objects 4-1

About Color in QuickDraw GX	4-5
Color Spaces	4-6
Luminance-Based Color Spaces	4-7
RGB-Based Color Spaces	4-9
CMYK Color Spaces	4-14
Universal Color Spaces	4-15
Indexed Color Spaces	4-22
Color Spaces With Alpha Channels	4-24
Color-Component Values, Color Values, and Colors	4-25
Color Conversion and Color Matching	4-26
Color Profiles	4-28
Color-Matching Methods	4-30
When Color Matching Occurs	4-31
About Color Set Objects	4-32
Color Set Properties	4-33
Color Values in a Color Set	4-34
Default Color Sets	4-34
About Color Profile Objects	4-35
Color Profile Properties	4-36
Profile Data	4-36
The Default Color Profile	4-37
Zero-Length Profiles	4-37
Using Colors and Color-Related Objects	4-38
Assigning Colors to Shapes	4-38
Assigning Color Profiles to Colors	4-39
Comparing and Testing Colors	4-40
Checking for Out-of-Gamut Colors	4-40
Checking Colors for Closeness and Color Space	4-40

Predicting Drawing Results	4-41
Converting and Matching Colors	4-41
Creating and Manipulating Color Set and Color Profile Objects	4-42
Creating and Disposing of a Color Set or Color Profile	4-42
Copying, Comparing, and Cloning Color Sets and Color Profiles	4-44
Loading and Unloading Color Sets and Color Profiles	4-45
Manipulating Object Properties of Color Sets and Color Profiles	4-46
Manipulating Owner Counts	4-46
Getting and Setting Tag References	4-47
Manipulating the Colors in a Color Set Object	4-47
Manipulating the Profile Data in a Color Profile Object	4-48
Colors and Color-Related Objects Reference	4-49
Constants and Data Types	4-50
Color-Component Values	4-50
Color Values	4-50
The Color Structure	4-53
Color Packing	4-54
Color Spaces	4-55
The Color Set Object	4-56
The gxSetColor Union	4-56
The Color Profile Object	4-57
Color Functions	4-57
Color Set Functions	4-62
Creating and Manipulating Color Set Objects	4-62
Manipulating Color Set Object Properties	4-69
Retrieving and Replacing Colors in a Color Set	4-73
Color Profile Functions	4-78
Creating and Manipulating Color Profile Objects	4-78
Manipulating Color Profile Object Properties	4-84
Retrieving and Replacing Profile Information	4-88
Summary of Colors and Color-Related Objects	4-94
Constants and Data Types	4-94
Color Functions	4-98
Color Set Functions	4-98
Color Profile Functions	4-99

Chapter 5

Ink Objects 5-1

About Ink Objects	5-5
Ink Properties	5-6
Color	5-7
Transfer Mode	5-8
Ink Attributes	5-9
The Default Ink Object	5-10

About Transfer Modes	5-11
Transfer Mode Types	5-11
Arithmetic Transfer Modes	5-12
Highlight Transfer Mode	5-15
Boolean Transfer Modes	5-16
Pseudo-Boolean Transfer Modes	5-18
Alpha-Channel Transfer Modes	5-20
Transfer Mode Color Space	5-25
Color Limits	5-27
Source Color Limits	5-31
Destination Color Limits	5-32
Result Color Limits	5-32
Transfer Mode Matrices	5-33
Flags	5-34
Transfer Component Flags	5-35
Transfer Mode Flags	5-35
Summary of Transfer Mode Operation	5-36
Using Ink Objects	5-38
Creating and Manipulating Ink Objects	5-38
Creating and Disposing of Ink Objects	5-38
Copying, Comparing, and Cloning Ink Objects	5-39
Loading and Unloading Ink Objects	5-40
Manipulating Ink Object Properties	5-40
Getting and Setting an Ink Object's Attributes	5-40
Manipulating an Ink Object's Owner Count	5-41
Getting and Setting an Ink Object's Tag References	5-41
Getting and Setting an Ink Object's Color	5-42
Getting and Setting an Ink Object's Transfer Mode	5-43
Working With Transfer Modes	5-44
Simple Source-to-Destination Transfers	5-44
Drawing Selected Parts of the Source	5-45
Preserving Selected Parts of the Destination	5-45
Copying or Preserving Luminance	5-46
Modifying Luminance	5-47
Isolating and Modifying Color Ranges	5-47
Masking	5-48
Partial Transparency	5-48
Anti-Aliasing	5-49
Making Color Separations	5-49
Transfer Modes and Printing	5-49
Ink Objects Reference	5-50
Constants and Data Types	5-50
The Ink Object	5-50
Ink Attributes	5-51
Color Structure	5-51
Transfer Mode Structure	5-52
Transfer Mode Flags	5-53

Transfer Component Structure	5-53
Component Modes (Transfer Mode Types)	5-55
Transfer Component Flags	5-55
Functions	5-56
Creating and Manipulating Ink Objects	5-56
Manipulating Ink Object Properties	5-60
Getting and Setting an Ink's Color	5-68
Getting and Setting an Ink's Transfer Mode	5-72
Summary of Ink Objects	5-77
Constants and Data Types	5-77
Functions	5-79

Chapter 6 Transform Objects 6-1

About Transform Objects	6-5
Transform Object Properties	6-6
Clip	6-7
Mapping	6-10
View Port List	6-11
Hit-Test Parameters	6-11
Default Transform Objects	6-14
Using Transform Objects	6-15
Creating and Manipulating Transform Objects	6-15
Creating and Disposing of Transform Objects	6-15
Copying, Comparing, and Cloning Transform Objects	6-16
Implicit Creation of Transform Objects	6-18
Loading and Unloading Transform Objects	6-18
Manipulating Transform Object Properties	6-19
Manipulating a Transform Object's Owner Count	6-19
Getting and Setting a Transform Object's Tag References	6-20
Resetting Default Transform Properties	6-20
Getting, Setting, and Modifying the Transform Clip	6-20
Moving, Scaling, Rotating, and Skewing Shapes	6-23
Modifying the Transform Mapping	6-24
Modifying Shape Geometry	6-26
Manipulating the View Port List	6-28
Setting Up Hit-Test Parameters	6-30
Transform Objects Reference	6-31
Constants and Data Types	6-31
The Transform Object	6-31
Shape Parts for Hit-Testing	6-32
Functions	6-32
Creating and Manipulating Transform Objects	6-33
Manipulating Transform Object Properties	6-38
Getting and Setting the Clip	6-43
Performing Geometric Operations on Transform Clips	6-48

Getting and Setting the Mapping	6-53
Transforming Shapes by Modifying Transform Mappings	6-58
Transforming Shapes by Modifying Shape Geometries	6-65
Getting and Setting the View Port List	6-73
Getting and Setting the Hit-Test Parameters	6-77
Summary of Transform Objects	6-82
Constants and Data Types	6-82
Functions	6-83

Chapter 7 View-Related Objects 7-1

About View Ports, View Devices, and View Groups	7-5
About View Port Objects	7-7
View Port Properties	7-7
View Port Clip and Mapping	7-9
Dither	7-10
Halftone	7-13
Parent and Child View Ports	7-18
View Port Attributes	7-20
The Default View Port Object	7-20
View Port Objects and Windows	7-21
About View Device Objects	7-24
View Device Properties	7-25
View Device Clip and Mapping	7-26
View Device Bitmap	7-26
View Device Attributes	7-27
The Default View Device Object	7-28
View Device Objects and Physical Devices	7-28
About View Group Objects	7-29
View Groups Have No Properties	7-29
Onscreen and Offscreen View Groups	7-29
About Drawing, Coordinate Conversion, and Clipping	7-30
QuickDraw GX Coordinates	7-31
Geometry Space	7-32
Local Space	7-33
Global Space	7-34
Device Space	7-38
Using View-Related Objects	7-39
Using View Ports	7-40
Creating and Manipulating View Port Objects	7-40
Manipulating View Port Object Properties	7-42
Getting and Setting a View Port's Clip and Mapping	7-44
Setting Up the View Port Hierarchy for a Window	7-46
Supporting Scrolling in a Window	7-47
Identifying a View Port's View Devices	7-49
Identifying a Shape's View Ports	7-50

Measuring a Shape in Local Space	7-51
Using View Devices	7-52
Creating and Manipulating View Device Objects	7-52
Manipulating View Device Object Properties	7-54
Getting and Setting a View Device's Clip and Mapping	7-56
Identifying a Shape's View Devices	7-58
Measuring a Shape in Device Space	7-59
Hit-Testing a Shape on a Device	7-60
Using View Groups	7-60
Creating and Manipulating View Group Objects	7-61
Setting Up an Offscreen View Group	7-62
Measuring a Shape in Global Space	7-63
View-Related Objects Reference	7-65
Constants and Data Types	7-65
The View Port Object	7-65
The Halftone Structure	7-65
Dot Types	7-66
Tint Types	7-67
View Port Attributes	7-68
The View Device Object	7-68
View Device Attributes	7-68
The View Group Object	7-69
View Group Types	7-69
View Port Functions	7-69
Creating and Manipulating View Port Objects	7-70
Manipulating View Port Object Properties	7-74
Retrieving the View Devices That Intersect a View Port	7-94
Retrieving the View Ports That Intersect a Shape	7-95
Measuring a Shape in Local Coordinates	7-96
View Device Functions	7-97
Creating and Manipulating View Device Objects	7-97
Manipulating View Device Object Properties	7-102
Retrieving the View Devices That Intersect a Shape	7-115
Measuring a Shape in Device Coordinates	7-116
Measuring the Colors and Pattern Width of a Shape on a Device	7-118
Hit-Testing a Shape on a Device	7-120
View Group Functions	7-121
Creating and Disposing of View Group Objects	7-121
Getting the View Ports and View Devices of a View Group	7-123
Measuring a Shape in Global Coordinates	7-125
Summary of View-Related Objects	7-127
Constants and Data Types	7-127
View Port Functions	7-129
View Device Functions	7-130
View Group Functions	7-131

About Tag Objects	8-3
Tag Object Properties	8-4
Tag Types	8-5
Uses for Tag Objects	8-6
Using Tag Objects	8-7
Creating and Manipulating Tag Objects	8-7
Creating and Deleting a Tag Object	8-8
Copying, Comparing, and Cloning Tag Objects	8-9
Loading and Unloading Tag Objects	8-9
Manipulating Tag Object Properties	8-9
Getting and Setting a Tag Object's Tag Type and Contents	8-10
Manipulating a Tag Object's Owner Count	8-11
Directly Manipulating Tag Object Contents	8-11
Attaching Tags to a QuickDraw GX Object	8-12
Tag Objects Reference	8-12
Constants and Data Types	8-13
The Tag Object	8-13
Functions	8-13
Creating and Manipulating Tag Objects	8-13
Manipulating Tag Object Properties	8-18
Directly Manipulating the Data in a Tag Object	8-21
Summary of Tag Objects	8-25
C Summary	8-25
Functions	8-25

Figures, Tables, and Listings

Color Plates

Color plates are immediately preceding the title page

Color Plate 1	Blend example with different operand values
Color Plate 2	Showing color transparency with an alpha channel
Color Plate 3	Applying color by preserving luminance in the destination
Color Plate 4	Color spaces

Preface

About This Book xvii

Figure P-1	Roadmap to the QuickDraw GX suite of books	xx
-------------------	--	----

Chapter 1

Introduction to QuickDraw GX 1-1

Figure 1-1	Several QuickDraw GX objects	1-8
Figure 1-2	A shape object and its referenced objects	1-12
Figure 1-3	Printing objects	1-14
Figure 1-4	Effects of mapping	1-25
Figure 1-5	How QuickDraw GX draws a shape	1-27
Figure 1-6	A rectangle in geometry space	1-29
Figure 1-7	A rectangle in local space (transform mapping applied)	1-30
Figure 1-8	A rectangle in global space (view port mapping applied)	1-31
Figure 1-9	A rectangle in device space (view device mapping applied)	1-32
Figure 1-10	Parts of a line for hit-testing	1-33
Figure 1-11	Dragging a document to a desktop printer icon on the desktop	1-36
Figure 1-12	Printing a single document that has multiple formats	1-37
Figure 1-13	Properties of the basic QuickDraw GX objects	1-49
Table 1-1	Convenience constants for parameters	1-43
Table 1-2	QuickDraw GX objects	1-45
Listing 1-1	Sample GraphicsBug heap dump (HD) listing	1-40

Chapter 2

Shape Objects 2-1

Figure 2-1	Basic components of a QuickDraw GX shape	2-6
Figure 2-2	The shape object and its properties	2-7
Figure 2-3	Shape geometry for each type of QuickDraw GX shape	2-12
Figure 2-4	Even-odd and winding fills	2-14
Figure 2-5	Shape parts for hit-testing	2-21
Table 2-1	Shape types	2-9
Table 2-2	Shape fills	2-13

Table 2-3	Valid shape fills for each shape type	2-15
Table 2-4	Shape attributes	2-16
Table 2-5	Where to find information on shape-type conversion	2-33
Table 2-6	Shape-related functions described elsewhere	2-42
Listing 2-1	Directly accessing a shape's geometry	2-34
Listing 2-2	Hit-testing a line	2-38
Listing 2-3	Flattening a shape	2-39
Listing 2-4	Unflattening a shape	2-40
Listing 2-5	A spool function that parses shape data	2-41

Chapter 3

Style Objects 3-1

Figure 3-1	The style object and its properties	3-4
Table 3-1	Where to go for information on style object properties and functions	3-6
Table 3-2	Style-related functions described elsewhere	3-14
Listing 3-1	Building a style list by copying a style object	3-9

Chapter 4

Colors and Color-Related Objects 4-1

Figure 4-1	Luminance color space	4-7
Figure 4-2	Storage formats for luminance-based color spaces	4-8
Figure 4-3	RGB color space	4-9
Figure 4-4	Storage formats for RGB color spaces	4-11
Figure 4-5	HSV color space and HLS color space	4-12
Figure 4-6	Storage formats for HSV color spaces	4-13
Figure 4-7	Colors in CMYK color space	4-14
Figure 4-8	Storage formats for CMYK color spaces	4-15
Figure 4-9	Yxy chromaticities	4-17
Figure 4-10	Storage formats for XYZ color spaces	4-20
Figure 4-11	The I and Q axes in YIQ color space	4-21
Figure 4-12	Storage formats for YIQ color spaces	4-22
Figure 4-13	Storage format for indexed color space	4-23
Figure 4-14	Showing color transparency with an alpha channel	4-24
Figure 4-15	Color gamuts for two devices (in Yxy space)	4-28
Figure 4-16	Profile chromaticities for a device (in Yxy space)	4-29
Figure 4-17	A profile response curve for a device	4-29
Figure 4-18	Maintaining lightness and maintaining saturation in color matching	4-31
Figure 4-19	The color set object and its properties	4-33
Figure 4-20	The color profile object and its properties	4-36
Table 4-1	Luminance-based color spaces supported by QuickDraw GX	4-8
Table 4-2	RGB color spaces supported by QuickDraw GX	4-10
Table 4-3	HSV and HLS color spaces supported by QuickDraw GX	4-13
Table 4-4	CMYK color spaces supported by QuickDraw GX	4-15

Table 4-5	Universal color spaces supported by QuickDraw GX	4-19
Table 4-6	Video color spaces supported by QuickDraw GX	4-21
Table 4-7	Indexed color space supported by QuickDraw GX	4-23

Chapter 5

Ink Objects 5-1

Figure 5-1	The ink object and its properties	5-6
Figure 5-2	Arithmetic transfer modes	5-13
Figure 5-3	Blend example with different operand values	5-15
Figure 5-4	Highlight transfer mode	5-16
Figure 5-5	Boolean transfer modes (1-bit depth)	5-17
Figure 5-6	Pseudo-Boolean transfer modes	5-18
Figure 5-7	Alpha-channel transfer modes	5-21
Figure 5-8	Typical modes used to determine result opacity for the alpha channel	5-23
Figure 5-9	Anti-aliasing	5-25
Figure 5-10	Automatic conversion of color values during a transfer mode operation	5-26
Figure 5-11	Maximum and minimum color-component values in RGB space	5-28
Figure 5-12	How minimum and maximum color limits affect drawing	5-29
Figure 5-13	How reversed minimum and maximum color limits affect drawing	5-29
Figure 5-14	The effects of reversing maximum and minimum in a color space	5-30
Figure 5-15	The effect of source color limits on drawing	5-31
Figure 5-16	The effect of destination color limits on drawing	5-32
Figure 5-17	The effect of result color limits on drawing	5-33
Figure 5-18	Summary of transfer mode operation	5-37
Figure 5-19	Applying color by preserving luminance in the destination	5-47
Table 5-1	Ink attributes	5-9

Chapter 6

Transform Objects 6-1

Figure 6-1	The transform object and its properties	6-6
Figure 6-2	A transform clip	6-7
Figure 6-3	A framed transform clip	6-8
Figure 6-4	Converting a framed shape with a nonzero pen width into a transform clip	6-8
Figure 6-5	Using a bitmap as a transform clip	6-9
Figure 6-6	Modifying a transform clip by subtracting it from another shape	6-9
Figure 6-7	Effects of the transform mapping	6-10
Figure 6-8	Constructive geometry operations with a polygon clip and a rectangle shape	6-22
Table 6-1	Shape parts for hit-testing, from the gxShapeParts enumeration	6-12
Table 6-2	Constructive geometry operations between transform clips and other shapes	6-21

Listing 6-1	Creating and disposing of a transform object	6-16
Listing 6-2	Cloning a transform to prevent it from being deleted	6-17
Listing 6-3	Modifying a shape's transform with transform-mapping calls only	6-25
Listing 6-4	Modifying a shape's transform with transform-mapping and shape-geometry calls	6-25
Listing 6-5	Modifying a shape's geometry with shape-geometry calls	6-27
Listing 6-6	Getting and setting view ports	6-29

Chapter 7

View-Related Objects 7-1

Figure 7-1	Objects used by the drawing mechanism	7-6
Figure 7-2	View port object properties	7-8
Figure 7-3	Clipping and mapping in view ports	7-10
Figure 7-4	Halftone angle	7-14
Figure 7-5	Halftone frequency	7-15
Figure 7-6	Halftone dot types	7-16
Figure 7-7	Hierarchical view ports in a window	7-18
Figure 7-8	A view port hierarchy	7-19
Figure 7-9	View ports in windows	7-22
Figure 7-10	Adjusting a child view port's mapping to handle scrolling	7-23
Figure 7-11	View ports overlapping view devices	7-24
Figure 7-12	View device object properties	7-25
Figure 7-13	The QuickDraw GX coordinate plane	7-31
Figure 7-14	A shape geometry and a transform clip geometry	7-32
Figure 7-15	Applying the transform's clip and mapping to a shape	7-33
Figure 7-16	Applying the child view port's mapping and clip to a shape	7-35
Figure 7-17	Applying the parent view port's mapping and clip to a shape	7-36
Figure 7-18	Applying the view device's mapping and clip to a shape	7-38
Figure 7-19	The shape as finally displayed	7-39
Table 7-1	Dither levels and patterns	7-11
Table 7-2	View port attributes	7-20
Table 7-3	View device attributes	7-27
Listing 7-1	Changing a view port's dither, halftone, and attributes	7-42
Listing 7-2	Copying the view ports from one view group to another	7-44
Listing 7-3	Changing a view port's mapping	7-45
Listing 7-4	Setting a view port clip	7-46
Listing 7-5	Setting up a view port for a window	7-47
Listing 7-6	Supporting scrolling in a child view port	7-48
Listing 7-7	Setting a shape color for XOR highlighting	7-49
Listing 7-8	Locating the bounding rectangles of a list of shapes in a view port	7-51
Listing 7-9	Creating a new view device	7-53
Listing 7-10	Copying the view devices from one view group to another	7-54
Listing 7-11	Returning the mapping from local to device space	7-57
Listing 7-12	Setting up a data structure for offscreen drawing	7-58
Listing 7-13	Setting up a data structure for offscreen drawing	7-61

Listing 7-14	Setting up a view port and view group for offscreen drawing	7-63
Listing 7-15	Returning the characteristics of an offscreen device area	7-64

Chapter 8

Tag Objects	8-1	
<hr/>		
Figure 8-1	The tag object and its properties	8-4
Table 8-1	Defined tag types for tag objects	8-6
Listing 8-1	Adding data to a shape as a tag object	8-8
Listing 8-2	Retrieving the contents of a tag object	8-10

About This Book

QuickDraw GX is an integrated, object-based approach to graphics programming on Macintosh computers. This book, *Inside Macintosh: QuickDraw GX Objects*, gets you started by describing the object system and showing you how to create and manipulate the fundamental QuickDraw GX objects.

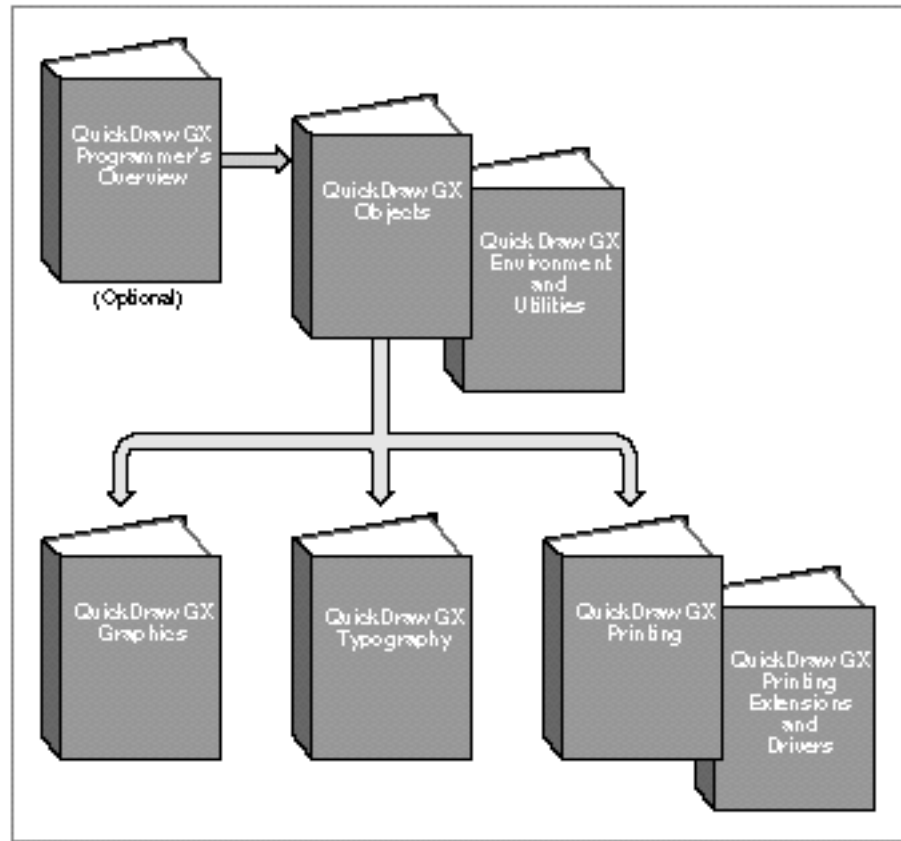
For application programming purposes, QuickDraw GX augments the capabilities of some of the Macintosh system software managers documented in other parts of *Inside Macintosh*. In situations where your application uses QuickDraw GX for drawing, information in this book replaces much of the information in *Inside Macintosh: Imaging With QuickDraw*. However, QuickDraw and QuickDraw GX coexist without conflict, and you can use both within the same program. Furthermore, for tasks outside the scope of QuickDraw GX, such as managing cursors or hardware color tables, you need to use QuickDraw.

Before you read this book, you should already be familiar with the Macintosh Toolbox, as described in *Inside Macintosh: Macintosh Toolbox Essentials* and *Inside Macintosh: More Macintosh Toolbox*. See the inside back cover of this book for a diagram showing those books and the others that make up the *Inside Macintosh* suite.

This book is the first reference book in the *Inside Macintosh QuickDraw GX* suite; read it before reading other references, such as *Inside Macintosh: QuickDraw GX Graphics* and *Inside Macintosh: QuickDraw GX Typography*. Figure P-1 shows the suggested reading order for the QuickDraw GX books.

For an alternative approach to learning QuickDraw GX, you can read *QuickDraw GX Programmer's Overview* before or along with this book. *QuickDraw GX Programmer's Overview* teaches QuickDraw GX programming through building extensive code samples.

Figure P-1 Roadmap to the QuickDraw GX suite of books



What to Read

This book is for all QuickDraw GX programmers. You can read the chapters in any order, except that the first chapter introduces concepts that the others build on:

- n Chapter 1, “Introduction to QuickDraw GX,” provides an overview of all of QuickDraw GX, concentrating especially on its capabilities for managing and drawing objects. Read this chapter first.
- n Chapter 2, “Shape Objects,” describes how to create and use QuickDraw GX shapes, the basic objects that you draw. (To apply shape objects to specific graphic and typographic tasks, the chapter refers you to the books *Inside Macintosh: QuickDraw GX Graphics* and *Inside Macintosh: QuickDraw GX Typography*, respectively.)

P R E F A C E

- n Chapter 3, “Style Objects,” describes how to create and use QuickDraw GX style objects, whose purpose is to modify the appearance or behavior of shape objects. (To apply style objects to specific graphic and typographic tasks, the chapter refers you to the books *Inside Macintosh: QuickDraw GX Graphics* and *Inside Macintosh: QuickDraw GX Typography*, respectively.)
- n Chapter 4, “Colors and Color-Related Objects,” describes the QuickDraw GX approach to color representation, and the objects that contain color information. This chapter describes how to create and use color set objects, which are used to implement indexed color spaces, and color profile objects, which are used for color matching.
- n Chapter 5, “Ink Objects,” describes how to create and use QuickDraw GX ink objects, which specify the color and transfer mode used to draw a shape.
- n Chapter 6, “Transform Objects,” describes how to create and use QuickDraw GX transform objects, which are used to position and transform the appearance of a shape, and to store information for hit-testing.
- n Chapter 7, “View-Related Objects,” describes how to create and use view ports, view devices, and view groups, which are QuickDraw GX objects that work together to provide flexible capabilities in onscreen and offscreen drawing.
- n Chapter 8, “Tag Objects,” describes how to create and use QuickDraw GX tag objects, which can contain any kind of information that you can use in any way to extend the capabilities of other QuickDraw GX objects.

Other kinds of QuickDraw GX objects are described in other books. See the chapter “Introduction to QuickDraw GX” for information and cross-references.

The color plate at the front of this book shows full-color examples of some of the figures found elsewhere in the book, in the chapters “Colors and Color-Related Objects” and “Ink Objects.”

Chapter Organization

Most chapters in this book follow a standard general structure. For example, the chapter “Transform Objects” contains these major sections:

- n “About Transform Objects.” This section provides an overview of transform objects.
- n “Using Transform Objects.” This section describes how you can create and manipulate transform objects using QuickDraw GX. It describes how to use the most common functions, gives related user interface information, provides code samples, and supplies additional information.

- n “Transform Object Reference.” This section provides a complete reference for transform objects by describing the constants, data types, and functions that you use with transform objects. Each function description follows a standard format, which gives the function declaration; a description of every parameter; the function result, if any; and a list of errors, warnings, and notices. Most function descriptions give additional information about using the function and include cross-references to related information elsewhere.
- n “Summary of Transform Objects.” This shows the C interface for the constants, data types, and functions associated with transform objects.

Conventions Used in This Book

This book uses various conventions to present certain types of information.

Special Fonts

All code listings, reserved words, and the names of data structures, constants, fields, parameters, and functions are shown in `Courier` (`this is Courier`).

When new terms are introduced, they are in **boldface**. These terms are also defined in the glossary.

Types of Notes

There are several types of notes used in this book.

Note

A note formatted like this contains information that is interesting but possibly not essential to an understanding of the main text. The wording in the title may say something more descriptive than just “Note,” for example “Terminology Note.” (An example appears on page 1-4.) u

IMPORTANT

A note like this contains information that is especially important. (An example appears on page 2-35.) s

Numerical Formats

Hexadecimal numbers are shown in this format: `0x0008`.

The numerical values of constants are shown in decimal, unless the constants are flag or mask elements that can be summed, in which case they are shown in hexadecimal.

Type Definitions for Enumerations

Enumeration declarations in this book are commonly followed by a type definition that is not strictly part of the enumeration. You can use the type to specify one of the enumerated values for a parameter or field. The type name is usually the singular of the enumeration name, as in the following example:

```
enum gxDashAttributes {
    gxBendDash          = 0x0001,
    gxBreakDash         = 0x0002,
    gxClipDash          = 0x0004,
    gxLevelDash         = 0x0008,
    gxAutoAdvanceDash = 0x0010
};
typedef long gxDashAttribute;
```

Illustrations

This book uses several conventions in its illustrations.

In illustrations that show object properties, properties that are object references are in italics. See, for example, Figure 2-2 in Chapter 2.

Objects in diagrams, whether shown with their properties or without, are represented by distinctive icons, such as these:



See, for example, Figure 1-1 in Chapter 1 and Figure 2-1 in Chapter 2.

Development Environment

The QuickDraw GX functions described in this book are available using C interfaces. How you access these functions depends on the development environment you are using.

Code listings in this book are shown in ANSI C. They suggest methods of using various functions and illustrate techniques for accomplishing particular tasks. Although most code listings have been compiled and tested, Apple Computer, Inc., does not intend for you to use these code samples in your applications.

Developer Products and Support

APDA is Apple's worldwide source for over three hundred development tools, technical resources, training products, and information for anyone interested in developing applications on Apple platforms. Customers receive the quarterly *APDA Tools Catalog* featuring all current versions of Apple development tools and the most popular third-party development tools. Ordering is easy; there are no membership fees, and application forms are not required for most of our products. APDA offers convenient payment and shipping options, including site licensing.

To order products or to request a complimentary copy of the *APDA Tools Catalog*, contact

APDA
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone	800-282-2732 (United States) 800-637-0029 (Canada) 716-871-6555 (International)
Fax	716-871-6511
AppleLink	APDA
America Online	APDAorder
CompuServe	76666,2405
Internet	APDA@applelink.apple.com

If you provide commercial products and services, call 408-974-4897 for information on the developer support programs available from Apple.

Introduction to QuickDraw GX

Contents

What Is QuickDraw GX?	1-3
Color Graphics	1-4
Typography	1-5
Printing	1-6
What QuickDraw GX Is Not	1-7
QuickDraw GX Objects	1-7
How QuickDraw GX Defines Objects	1-8
Advantages of an Object-Based Structure	1-9
Kinds of QuickDraw GX Objects	1-10
Shape Objects	1-10
Supporting Objects	1-11
Printing Objects	1-14
Object Properties	1-15
Default Objects and Default Properties	1-17
Adding Custom Behavior With Tag Objects	1-17
Objects and Memory	1-18
Application Memory and QuickDraw GX Memory	1-18
Sharing and Multiple Object References	1-19
Owner Count	1-20
Cloning	1-20
Automatic Loading and Unloading of Objects	1-21
Direct Access to Object Structure: Locking and Unlocking	1-22
External Storage of Objects: Flattening and Unflattening	1-23

CHAPTER 1

Drawing and Hit-Testing Shapes	1-23
Drawing	1-24
Mapping and Clipping	1-24
View-Related Objects	1-25
The Drawing Sequence: Coordinate Conversion	1-28
Hit-Testing	1-32
Printing With QuickDraw GX	1-34
Core Printing Features	1-35
Custom Dialog Boxes and Page Formats	1-36
Advanced Printing Features	1-37
The QuickDraw GX Programming Environment	1-38
Setting Up QuickDraw GX Memory	1-38
Handling Errors	1-38
Debugging	1-39
Debugging and Non-Debugging Versions	1-39
Debugging With GraphicsBug	1-40
Programming Conventions and Consistencies	1-41
Object Behavior	1-41
Functions and Function Results	1-41
Function Parameters	1-42
Code Naming Conventions	1-44
Relationship to the Macintosh Toolbox	1-44
Summary Table and Diagram of QuickDraw GX Objects	1-45

This chapter introduces the QuickDraw GX object-based approach to graphics programming. Any QuickDraw GX programming you do requires a basic understanding of objects and how to manipulate them. Read this chapter before reading any other chapter in this book, and before reading subsequent books in the QuickDraw GX suite, such as *Inside Macintosh: QuickDraw GX Graphics*, *Inside Macintosh: QuickDraw GX Typography*, and *Inside Macintosh: QuickDraw GX Printing*.

You can also start learning about QuickDraw GX by reading the book *QuickDraw GX Programmer's Overview*, either before or in conjunction with reading this chapter and the rest of the QuickDraw GX suite. *QuickDraw GX Programmer's Overview* introduces you to QuickDraw GX concepts through designing and building code samples.

This chapter starts by outlining the features and advantages of QuickDraw GX. It then describes

- n the kinds of objects defined by QuickDraw GX
- n how your application interacts with QuickDraw GX memory
- n how to use objects to draw and hit-test shapes
- n how to use objects to print documents
- n how to program within the QuickDraw GX environment

The chapter concludes with a table and diagram summarizing QuickDraw GX objects and their properties.

What Is QuickDraw GX?

QuickDraw GX is a programming environment and toolbox for powerful two-dimensional color graphics programming. QuickDraw GX helps you create graphic and typographic objects and display them on a variety of imaging devices, including printers. The QuickDraw GX software architecture is based on objects and is compatible with, but does not require, object-oriented programming techniques.

QuickDraw GX is a large system that provides many benefits. The rest of this section summarizes some of those benefits, in terms of its three principal areas of application: color graphics, typography, and printing.

Color Graphics

QuickDraw GX is a powerful graphics engine with integrated color support, a wide range of graphics primitives, and sophisticated modes of drawing. It can manipulate images in quite general ways, leading to many useful special effects. Highlights of the graphics capabilities of QuickDraw GX include the following:

- n Multiple types of graphic shapes. QuickDraw GX directly supports geometric shapes (points, lines, rectangles, polygons, curves, and paths), bitmap shapes, and picture shapes (shapes that are collections of other shapes).
- n Multiple types of typographic shapes. QuickDraw GX directly supports text shapes, glyph shapes, and layout shapes, which range from simple unstyled lines of text to multilanguage, multifont text lines with sophisticated typographic features.
- n Device independence. All positions and measurements in QuickDraw GX are independent of the resolution of any imaging device.
- n Flexible and powerful transformations. QuickDraw GX uses mathematical mappings to easily manipulate positions, dimensions, and distortions of shapes.
- n Easy stylistic variations. QuickDraw GX gives you great flexibility in setting shape characteristics such as pen width, patterns, font, and text face.
- n Device-independent colors. All colors in QuickDraw GX can be defined in a device-independent way and then converted to device-specific colors on any device.
- n Direct support for many color spaces, including luminance (for grayscale), RGB (for monitors), YIQ (for color video broadcast), CMYK (for printing), CIE and related device-independent color spaces (for colorimetrics).
- n Automatic color matching. QuickDraw GX automatically uses color profiles and the Macintosh ColorSync utilities to guarantee that a document's colors as displayed on a monitor match as closely as possible a printed copy of the same document. If you need to, you can also manually control the color matching process.
- n A sophisticated yet straightforward rendering mechanism. The mechanism allows multiple simultaneous views of a single shape, with different scales and orientations, on single or multiple devices, with simultaneous updating of all views if the shape is edited.

Compatibility With QuickDraw

QuickDraw GX does not replace the original QuickDraw architecture built into the Macintosh toolbox. An application that is not QuickDraw GX-aware is unaffected if QuickDraw GX is installed on the system. A QuickDraw GX application can also use standard QuickDraw calls and convert QuickDraw picture files into QuickDraw GX shapes. See the Macintosh environment chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities* for more information. u

The color graphics capabilities of QuickDraw GX are described both in this book and in *Inside Macintosh: QuickDraw GX Graphics*.

Typography

QuickDraw GX treats text both as text (a sequence of character codes that can be displayed and edited) and as graphics, meaning that all of the color graphics capabilities of QuickDraw GX are available for the display of text.

Each line of text can be a shape in QuickDraw GX. Using the typographic features of QuickDraw GX, you can generate and manipulate fully editable, text-related shapes with characteristics such as the following:

- n **Simplicity.** Text can have a single font at a single size, with no changes in stylistic variation along the line. This type of text is most useful in dialog boxes or other situations where relatively unsophisticated string presentation is needed.
- n **Flexible alignment and justification.** Text can be (1) left aligned, right aligned, or any point of alignment in between (including centered); and (2) unjustified, fully justified, or any level of justification in between.
- n **Multiple styles.** Each glyph or any set of glyphs can be styled (given a font, size, or set of typographic characteristics) independently of every other glyph.
- n **Independent glyph positions.** Each glyph can have any style and be positioned independently of every other glyph, so that text can be made to follow a curved path or circle.
- n **Sophisticated layout.** Text lines can exhibit great typographic sophistication, with features such as kerning, tracking, shifting, ligature formation, and contextual glyph substitution.
- n **Multilanguage text handling.** Text can be properly formatted in any language supported by a QuickDraw GX font, even contextual right-to-left languages such as Arabic, or languages with large character sets such as Chinese. Multiple languages, even with mixed text directions, can coexist on the same line.
- n **Vertical text.** Text such as Japanese and Chinese can be written vertically, and intermixed with properly oriented vertical Roman text.

Because a line of text is a QuickDraw GX shape, you can color it, fill it with a pattern, scale it, rotate it, and transform it like any graphic shape—all the while maintaining its identity and editability as a text line. You can also use certain typographic shapes, either as-is or converted to purely geometric shapes, to perform further graphic operations with them, such as clipping, dashing, and patterning.

QuickDraw GX also provides functions that help you manipulate sets of text lines, even the most typographically sophisticated text lines, for word-processing tasks such as hit-testing and line-breaking.

Much of the text-layout sophistication of QuickDraw GX depends on information in tables in QuickDraw GX fonts, which have many features—some of which may be enabled by default—that your application can use or disable, as desired.

The typographic capabilities of QuickDraw GX are described in detail in *Inside Macintosh: QuickDraw GX Typography*.

Printing

QuickDraw GX includes an extensible, device-independent printing architecture that provides a high level of support for both users and application developers, and that makes creation of printing extensions and printer drivers fast and efficient. The printing features of QuickDraw GX include the following:

- n A consistent application printing interface, regardless of the type of printer used.
- n A message-based printing system. Drivers, extensions, and even applications need only respond to (override) a standard set of printing messages if they wish to add specific functionality.
- n A set of printing objects that controls the printing process. Use of multiple objects means that, for example, different parts of a document can print on several printers simultaneously, or a single document can have multiple page formats for printing.
- n Support for desktop printers, which are represented by icons on the computer desktop. The user can print a document by dragging it to the icon. Desktop printers support printer sharing, and you can control jobs in the print queue of a desktop printer.
- n Customizable printing dialog boxes. In addition to standard print options, these dialog boxes also provide additional controls such as the ability to select a paper tray.
- n The capability of creating and reading portable digital documents (PDDs). These documents can be viewed or printed on any computer that has QuickDraw GX installed, without requiring the original application or fonts with which the document was created. If QuickDraw GX is installed, any application—including those that are not QuickDraw GX-aware—can create a PDD.
- n Fast development of printing extensions that can work with any printer driver and any application, to extend the printing capabilities available to the user.
- n Fast development of printer drivers.

To implement the basic printing capabilities of QuickDraw GX, your application need provide only a small amount of code, which executes in response to a few menu items and a single printing message. With additional developmental effort, you can provide highly customized capabilities. (Even if your application implements no QuickDraw GX printing features at all, its users receive the benefit of desktop printers.)

The application printing interface to QuickDraw GX is described in *Inside Macintosh: QuickDraw GX Printing*; the interface for printing extensions and printer drivers is described in *Inside Macintosh: QuickDraw GX Printing Extensions and Drivers*.

What QuickDraw GX Is Not

QuickDraw GX is a powerful system, but it does have certain limitations. If your graphic programming needs are outside of the capabilities of QuickDraw GX, you may wish to implement them yourself or—where possible—use the built-in capabilities of the platform on which your application runs. For example QuickDraw GX does not provide explicit support for

- n application-definable object methods
- n a floating-point interface
- n multiple colors or gradient fills in shapes (other than bitmaps)
- n planar-pixel devices
- n 3-D rendering
- n cubic curves (but conversion library code is available)
- n formatting of text units greater than a line, such as paragraphs
- n tabs in text
- n anti-aliasing (other than alpha-channel support in bitmaps)
- n palette management (handled by system software on the Macintosh)
- n cursor management (handled by system software on the Macintosh)

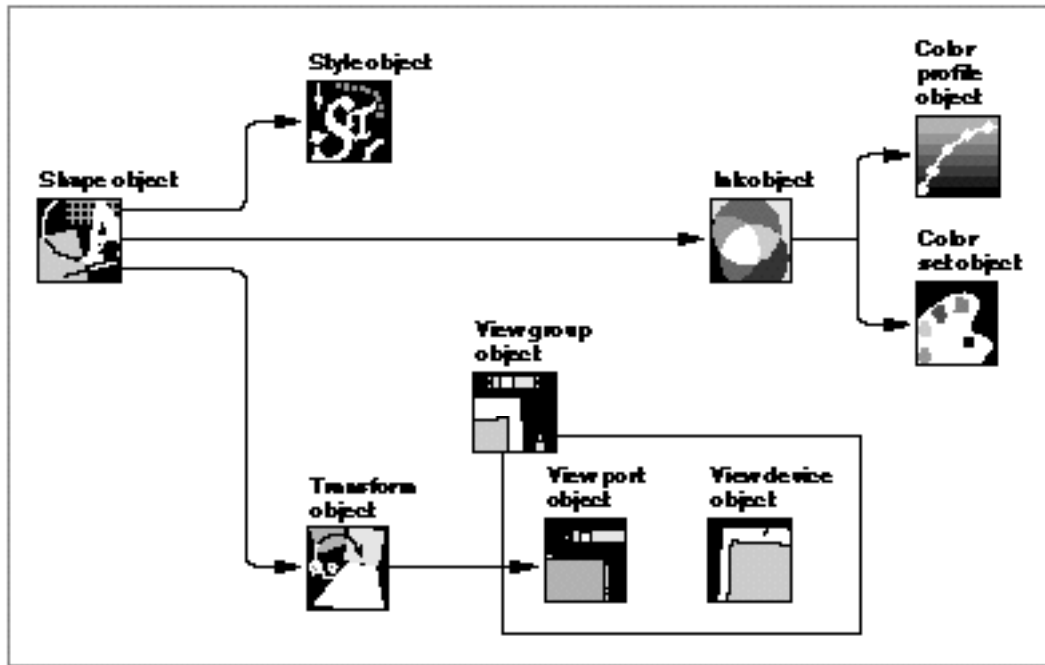
Also, on the Macintosh, QuickDraw GX does not completely replace either QuickDraw or the Window Manager for drawing, and it does not completely replace the Script Manager and international resources for non-Roman text-handling. QuickDraw GX extends the capabilities of these managers, but in some instances you still need to use their functions.

QuickDraw GX Objects

With QuickDraw GX you create and draw objects. Fundamental graphic shapes, such as rectangles and curves, are objects. Lines of text are objects. Other pieces of information, such as the color of a shape or the font used to draw a letter, are also kept in objects. Fonts are objects. Even the information that is used to describe the printing characteristics of a document is kept in objects.

Figure 1-1 names some of the common QuickDraw GX objects and shows their relationships, in terms of which objects use the information in which other objects. This chapter, this book, and much of the rest of the *Inside Macintosh QuickDraw GX* suite describe what these and other QuickDraw GX objects are and how to use them.

Figure 1-1 Several QuickDraw GX objects



How QuickDraw GX Defines Objects

Objects are specialized data structures. Some of the data structures used by operating systems such as the Macintosh Operating System are public—that is, your application can manipulate the values of their fields directly. Many of the data structures used by QuickDraw GX, on the other hand, are not public. These private data structures are called **objects** and the accessible pieces of information inside them are called **properties**. Your application creates and modifies objects to perform tasks, but it may not manipulate object properties directly. Instead, QuickDraw GX provides functions that manipulate them for you.

QuickDraw GX does not provide pointers or handles for you to locate objects. Instead, it provides reference values. To allow type checking in C and Pascal, QuickDraw GX defines references as pointers to structures, although the reference is *not* guaranteed to point to anything. For example, a shape object is identified by a shape **reference**:

```
typedef struct gxPrivateShapeRecord *gxShape;
```

The contents of the structure are private. To obtain information about an object, you must send its reference as a parameter to a QuickDraw GX function.

When you create an object, you call a `GXNewObject` function that returns a reference to the object. Conversely, you can dispose of an object you no longer need by passing its reference in a call to `GXDisposeObject`. For example, you can create a picture shape object by calling the `GXNewShape` function with a parameter that specifies that you want the shape to be a picture type:

```
myShape = GXNewShape(gxPictureType);
```

In this example, `myShape` is a reference to the shape object, returned by the function. When you are finished with the object, you dispose of it like this:

```
GXDisposeShape(myShape);
```

QuickDraw GX objects exist in a memory area (QuickDraw GX memory) that is separate from the application's memory. For more information on QuickDraw GX memory, see "Objects and Memory" beginning on page 1-18.

QuickDraw GX defines its objects in a device-independent manner. Because of that, and because many of its data structures are private, the QuickDraw GX software and the hardware on which it runs can evolve without disrupting existing applications.

Advantages of an Object-Based Structure

QuickDraw GX is currently implemented in the C programming language, which is not in itself object-oriented. Nevertheless, using QuickDraw GX gives you some of the fundamental programming advantages available with object-based systems.

QuickDraw GX objects are private. You do not usually have direct access to the internal data in a QuickDraw GX object; you instead make function calls to manipulate the information. This information hiding means that objects behave more consistently, unwanted side effects are minimized, and QuickDraw GX itself can take care of housekeeping tasks like tracking the current number of users of an object. It also means that QuickDraw GX can locate objects in memory managed by a graphics accelerator—memory that is not necessarily accessible to your application.

By analogy with the polymorphism of some object-oriented systems, QuickDraw GX functions are organized so that a single function can apply to many types of objects. For example, a single drawing command (`GXDrawShape`) draws any QuickDraw GX shape, from a point to a curve to a bitmap to a line of text. Furthermore, there are many classes of calls that, while defined individually for each kind of object they apply to (in order to facilitate type-checking in Pascal and C), are completely parallel in function and in syntax. For example, the functions `GXGetShapeTags`, `GXGetStyleTags`, and `GXGetInkTags` take the same parameter (an object reference) and perform the same task (return a list of associated tag objects), but each for a different kind of object.

QuickDraw GX objects can be shared. To save duplication and prevent the accumulation of excessive numbers of objects in memory, QuickDraw GX allows multiple references to a single object. QuickDraw GX tracks the number of references to an object. When you are finished with an object, you dispose of it; QuickDraw GX then makes sure that the object is not being used for any other purpose before actually deleting it from memory.

Creating a QuickDraw GX object is somewhat like instantiating a class in an object-oriented system. When you first create a QuickDraw GX object it typically has default values that you can use or change to suit your needs.

Object-manipulation functions are mostly consistent across all objects; categories include *GXNewObject* (makes a new object), *GXDisposeObject* (deletes the object), *GXCopyToObject* (copies an object), *GXEqualObject* (tests two objects for equality), and *GXCloneObject* (makes a shared reference). Object-editing functions are similarly consistent, and include *GXGetObjectProperty* (to retrieve values) and *GXSetObjectProperty* (to assign values). By combining *GXGetObject* and *GXSetObject* calls with index values and ranges, you can insert, delete, and replace all or parts of arrays of values within an object.

The QuickDraw GX environment provides other consistencies to make programming tasks more straightforward. Many are listed in the section “Programming Conventions and Consistencies” beginning on page 1-41.

Kinds of QuickDraw GX Objects

There are about a dozen different kinds of QuickDraw GX objects that you can use, beginning with the most fundamental object, the shape. Figure 1-1 on page 1-8 shows some of those objects and how they relate to each other; this section describes them and others.

Shape Objects

A **shape** is something that you can draw. Besides drawing it, you can also measure, parse, move, rotate, distort, check for intersection and union, make bold, simplify, and otherwise manipulate it. The fundamental purpose of QuickDraw GX is to create, manipulate, and draw shapes.

A shape consists of a **shape object** and three other associated objects (style, ink, and transform). A shape object consists of a **geometry** of a certain **shape type** (such as a line, rectangle, bitmap, or text) and information about how the geometry is framed or filled when drawn. A shape also has attributes, such as whether it should be stored in accelerator-card memory, if present. It also has references to its other three related objects.

Shapes and shape objects in general are discussed in the chapter “Shape Objects” in this book. More specifically, however, shapes are divided into types. There are two basic categories of shape type: graphic and typographic.

Graphic Shapes

Graphic shapes include geometric shapes, bitmap shapes, and picture shapes:

- n Geometric shapes are the building blocks for drawing. Geometric shapes, alone or in combination, make up the graphic elements supported by drawing programs. The defined types of geometric shapes are point, line, rectangle, curve, polygon, and path. There are two other special types of geometric shapes: empty and full. An empty shape has no extent, and a full shape has the maximum possible extent.
- n Bitmap shapes contain bit images or pixel images. QuickDraw GX bitmaps can be black and white, grayscale, or color.
- n Picture shapes are collections of other QuickDraw GX shapes. Picture shapes can contain other picture shapes, in a hierarchy. Picture shapes allow you to override some characteristics of the contained shapes.

Graphic shapes are described further in *Inside Macintosh: QuickDraw GX Graphics*. That book also describes functions for performing geometric operations, such as measurement, simplification, and constructive geometry, on graphic and typographic shapes.

Typographic Shapes

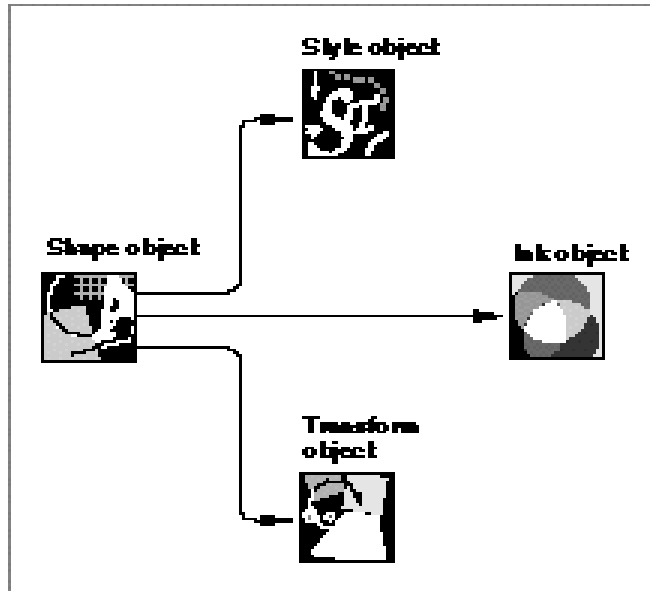
Typographic shapes represent text items—individual glyphs, collections of glyphs, or lines of text. The geometry of a typographic shape contains the text characters or glyphs of the shape, plus other information. There are three kinds of typographic shapes:

- n A text shape consists of a line of one or more characters or glyphs, all to be displayed in the same font with the same typestyle.
- n A glyph shape consists of one or more glyphs, each of which can be independently located, rotated, sized, and styled.
- n A layout shape consists of a line of text that can be in multiple languages, can have multiple writing directions (including vertical), can include ligatures and other contextual forms, and can display other sophisticated formatting and stylistic properties.

Typographic shapes are described further in *Inside Macintosh: QuickDraw GX Typography*.

Supporting Objects

Several other QuickDraw GX objects exist in support of shape objects. They are either directly or indirectly referenced by the shape object whose behavior they affect. Figure 1-2 shows the three objects that are directly referenced by a shape object; Figure 1-1 on page 1-8 includes these objects as well as additional objects referenced indirectly by the shape object.

Figure 1-2 A shape object and its referenced objects

Style Object

A **style object** describes certain characteristics affecting how a shape is drawn. For geometric shapes, this includes information such as the thickness of the pen, the joins between line segments, and any dash or pattern to apply to the shape. For typographic shapes, it includes information such as the font, text size, and typeface of the text. For layout shapes in particular, it includes information such as kerning behavior and font-feature selection.

Style objects in general are described in the chapter “Style Objects” in this book. Style objects used by graphic shapes are described in the geometric styles chapter of *Inside Macintosh: QuickDraw GX Graphics*; style objects used by typographic shapes are described in the typographic styles chapter of *Inside Macintosh: QuickDraw GX Typography*.

Ink Object

An **ink object** describes a shape’s color and its transfer mode—how that color is applied when the shape is drawn. Inks support many different kinds of color specification, and many different transfer modes.

Ink objects are described in the chapter “Ink Objects” in this book.

Transform Object

A **transform object** describes the clip and mapping applied to a shape when it is drawn. The clip limits the extent of the shape; it can be described by any shape geometry, and QuickDraw GX provides constructive geometry functions with which you can easily manipulate clips by combining them with other shapes. The mapping is a 3×3 matrix that defines translation, scaling, skewing, rotation, or perspective. Transforms also describe information used for hit-testing a shape and its parts. Transforms have references to one or more view ports, objects that describe where the shapes are drawn.

Transform objects are described in the chapter “Transform Objects” in this book.

Color Set Object and Color Profile Object

A **color set object** is like a color table; it contains an indexed set of colors. Color sets are used when colors are specified by index instead of by direct color value. Bitmaps commonly use color sets.

A **color profile object** contains color matching information. The information in a color profile can be used to convert device-specific colors to device-independent colors, to provide the most faithful reproduction of colors on different devices. QuickDraw GX can automatically perform color matching with available color profiles whenever it draws.

Color sets and color profiles are described in the chapter “Color and Color-Related Objects” in this book.

View Port Object, View Device Object, and View Group Object

A **view port object** is the location into which an application draws a shape. A view port object has a clip and a mapping that define a window (or a part of a window, such as a window pane). View ports can be arranged in a hierarchy.

A **view device object** typically describes a physical display device such as a monitor or printer (or an area of memory for offscreen drawing). It has a mapping, a clip, and a bitmap that describe the view device’s position, dimensions, pixel depth and colors, and color profile.

A **view group object** describes an imaging world, the global space in which view ports and view devices are located. Within a view group, view ports and view devices can overlap each other in any combination; the intersection of each view port with a view device determines what is actually drawn on that device.

View ports, view devices, and view groups are described in the chapter “View-Related Objects” in this book.

Tag Object

A **tag object** is a general container for information that an application wants to add to a QuickDraw GX object. Tag objects can have anything in them, from labels to alternate drawing instructions to anything else you feel is useful. You can attach a tag object to the tag list of most other kinds of objects (except other tag objects).

Tag objects are described in the chapter “Tag Objects” in this book.

Font Object

A **font object** is the QuickDraw GX representation of an installed font. A font object contains information about the font's names, encodings, font variations, and other tables. See the fonts chapter of *Inside Macintosh: QuickDraw GX Typography* for more information.

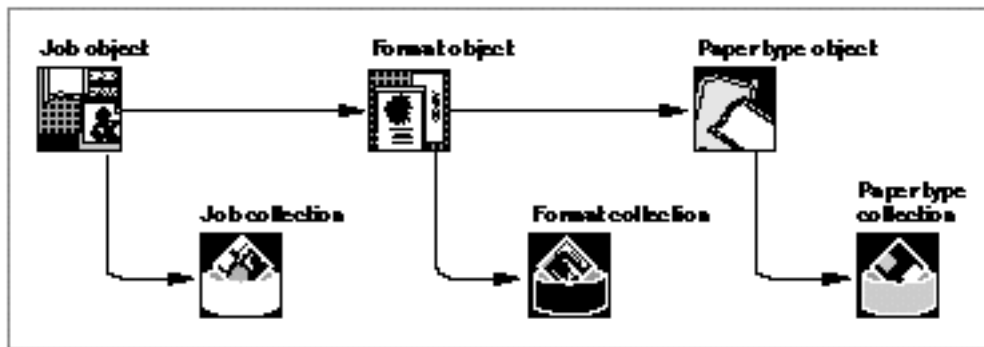
Graphics Client Object

A graphics client is the object representation of the QuickDraw GX memory allocated for an application, which is separate from the application's own memory. A graphics client has no accessible properties, and in most cases your application never explicitly creates one. See the memory management chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities* for more information.

Printing Objects

One category of QuickDraw GX objects exists to support printing. The printing objects include those shown in Figure 1-3 plus several others. Figure 1-3 shows the three principal QuickDraw GX printing objects (job, format, and paper-type), plus the three collection objects they use.

Figure 1-3 Printing objects



Note

Printing objects are different in some aspects from other QuickDraw GX objects. Most importantly, they exist in application memory instead of QuickDraw GX memory; this affects their behavior in several ways, as noted in later sections of this chapter. u

Job Object, Format Object, and Paper-Type Object

The **job object** is the primary holder of printing information for a document. Every printable document has a job object associated with it. The job object specifies information such as the number of copies and the page range for printing, and includes references to one or more format objects and two printer objects (one for formatting and one for current output).

The **format object** specifies information such as scaling and page dimensions for the document, and includes a reference to a paper-type object.

The **paper-type object** specifies information such as a paper-type name (such as “US Letter”) and the physical dimensions of the paper.

See the core printing features chapter of *Inside Macintosh: QuickDraw GX Printing* for more information.

Collection Objects

The job object, format object, and paper type object also include references to **collection objects**, which are objects managed by the Collection Manager, a part of system software provided with QuickDraw GX. Collection objects can contain any type of data; for printing, they hold additional useful information, such as specifications for halftoning, that is not in the printing objects. The Collection Manager is described in the Collection Manager chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

Printer Object

The **printer object** is another printing object. It represents a physical printer and includes a name and type, a driver name and type, and a reference to one or more view device objects that describe the characteristics of the printer the application draws to when printing. See the advanced printing features chapter of *Inside Macintosh: QuickDraw GX Printing* for more information.

Print-File Object

A **print-file object** is a printing object that represents a print file, the file that is the printable representation of a document. When it prints a document, QuickDraw GX first creates a print file, and then uses that print file to create an image on a printer. See the advanced printing features chapter of *Inside Macintosh: QuickDraw GX Printing* for more information.

Object Properties

The accessible information in an object is its set of properties. Object properties are like the fields of a structure, except that they are accessible only through function calls; you cannot read them directly. Each property consists of a value or a list of values; the definition of the property determines what it contains. For example, the shape type property of a shape object contains a value, such as `gxRectangleType`, that describes the type of shape that it is.

For most properties, QuickDraw GX provides `GXGetObjectProperty` and `GXSetObjectProperty` functions that allow you to get or set each accessible part of the object. For example, the following statement returns a shape object’s type into the `myShapeType` variable:

```
myShapeType = GXGetShapeType(myShape);
```

Figure 1-13 on page 1-49 lists the accessible properties of the principal QuickDraw GX objects other than printing objects. Note that, because they are properties and not fields, their order in Figure 1-13 is arbitrary. The properties are explained in more detail in the chapter that describes the object.

Some object properties are common to most kinds of objects. For example, many objects have properties that are simply references to other objects. In addition, many objects have attributes, an owner count, and a tag list. These four kinds of common properties are summarized in this section.

References

Some properties consist of references to other objects. These references define a relationship between the objects; the properties of the referenced object are like an extension to the properties of the object containing the reference. For example, Figure 1-2 on page 1-12 shows three objects referenced by a shape object: a style object, an ink object, and a transform object. Those three objects' properties affect how the shape that references them is drawn; the ink object, for example, defines the color of the shape.

Many objects contain references to other objects. Some object properties are individual references, whereas other properties are arrays, or lists, of references to several objects. The advantages of using object references are discussed in the section "Sharing and Multiple Object References" beginning on page 1-19.

Note

In illustrations of object properties throughout the QuickDraw GX documentation, properties that are object references (or lists of object references) are represented in italics. See, for example, how the style, ink and transform properties of the shape object are represented in Figure 1-13 on page 1-49. u

Attributes

Some objects have an attributes property, which is a group of flags that you use to modify the behavior of the object. In shapes, for example, these flags allow you to specify—among other things—how QuickDraw GX stores the shape object and how editing operations affect the shape object. In view ports, as another example, these flags allow you to specify behavior such as whether or not to perform color matching when drawing.

Owner Count

For objects that are shared, this property indicates how many references to the object exist. For example, when you create a new shape object, QuickDraw GX sets the owner count of the new shape to 1. If you add that shape to a picture, QuickDraw GX increments the shape's owner count by 1. If you dispose of the picture, QuickDraw GX decrements the shape's owner count by 1. Whenever the owner count of a shared object reaches 0, the object is deleted and its memory released.

Owner counts are discussed further in the section "Sharing and Multiple Object References" beginning on page 1-19.

Tag List

This property is an array of references to custom information stored in tag objects. Tag objects are discussed further in the section “Adding Custom Behavior With Tag Objects,” on this page.

Default Objects and Default Properties

QuickDraw GX provides default versions for all types of shape objects, and default values for the properties of other objects such as styles, inks, transforms, color sets, and color profiles. Therefore, when you create an object with a `GXNewObject` call, its properties are already set to match the default. For example, the default rectangle shape object has an owner count of 1, a solid shape fill, corners at locations (0.0, 0.0) and (0.0, 0.0), and a reference to the default ink object. If you want the new shape to have different dimensions or to reference a different ink object, you can change those properties after creating the shape.

The default shape objects are unique among QuickDraw GX default objects in that you can change them. If you want every new shape of a certain type to start off with a particular set of properties, you can change the properties of the default shape for that shape type, and every new shape of that type that you create will have the new properties.

You cannot change the default for most other objects. However, you can effectively change the default for any object that is referenced directly or indirectly by a shape object. For example, you can effectively create a new default ink object by first creating a version of the ink object that has the properties you want, and then altering all default shape objects to reference that ink object instead of the default ink object.

For objects for which there is no changeable default, there are nevertheless default values that are applied to the object when it is first created.

Default color sets and color profiles

Color sets have changeable default versions, but they function differently than default shapes. You can define a color set to be the default associated with bitmaps of a given pixel depth. However, when you create a color set using the `GXNewColorSet` function, it has specific properties that are unaffected by any previous definitions of defaults.

There is a single default color profile, applied by QuickDraw GX to colors that do not have an attached profile. The default profile is not directly changeable. ^u

Adding Custom Behavior With Tag Objects

A tag object is a special kind of object whose purpose is to allow any type of application-defined information to be attached to a QuickDraw GX object. An object such as a shape or transform can be “tagged” with data or code that provides extra information about it or allows you to alter its behavior in specific situations.

You can, for example, attach identifying strings to objects with tags. As another example, you can alter the way an object is displayed on a particular imaging device (such as a PostScript device) by attaching a tag to it that contains imaging commands specific to that device.

A tag object is attached to its associated object by means of a **tag list**, a property that most QuickDraw GX objects have. A tag list is an array of references to the tag objects attached to an object. Objects can thus have more than one attached tag object.

Because tags are QuickDraw GX objects, they can be shared. Like other QuickDraw GX objects, tags are accessible from objects in accelerator memory, they can be unloaded to disk and reloaded automatically, and they can be flattened (see “External Storage of Objects: Flattening and Unflattening” on page 1-23). See the chapter “Tag Objects” in this book for more information.

Objects and Memory

Objects are structures in memory. The way QuickDraw GX manages memory is central to its object orientation and to the advantages it provides you. QuickDraw GX has its own memory, and gives you access to it only in restricted situations.

Application Memory and QuickDraw GX Memory

When you program with QuickDraw GX, you are concerned with at least two separate memory heaps: the **application heap**, which holds your code and data structures, and a part of QuickDraw GX memory called the **graphics client heap**, which holds the objects you create with Quickdraw GX. As an application, you allocate variables and execute in application memory. You can directly access any data structures in that heap. Much of Macintosh system software, including the toolbox, can affect the application heap, sometimes in unwanted ways (as during memory compaction).

QuickDraw GX rarely uses the application heap (except for storing printing-related objects). It allocates its objects, structures, and variables in the graphics client heap. QuickDraw GX memory is private; you cannot directly access the contents of the graphics client heap except under special conditions. The graphics client heap does not even have to be in the same physical address space as the application heap. For example, QuickDraw GX can execute from and store objects in the memory on a graphics accelerator card.

QuickDraw GX objects are private because they are in private memory. That means you must make QuickDraw GX calls to access objects and their information, but it also means that you can make almost any call without worrying that it might move application memory.

Typically, your application manages its own structures in the application heap, and makes function calls to obtain or change the contents of the graphics client heap. For example when you call a `GXGetObjectProperty` function, QuickDraw GX places a copy of the contents of an object's property in your application's heap. If you modify the information, you can then call a `GXSetObjectProperty` function to copy the new values from your application's heap back into the object in the graphics client heap.

If you are a Macintosh programmer, remember that QuickDraw GX memory is completely separate, and you needn't be concerned about its location or contents. Macintosh Memory Manager functions cannot allocate, resize, or determine the size of any QuickDraw GX object. To manage its memory, QuickDraw GX has its own internal memory manager and memory management functions. See the memory management chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities* for more information. See *Inside Macintosh: Memory* for information on the Macintosh Memory Manager.

The QuickDraw GX memory manager may move objects, unload them to disk if necessary, and reload them when they are needed again. To reference and use an object, you needn't be concerned with or even know whether it is in a loaded or unloaded state. QuickDraw GX automatically loads any unloaded object when it is needed, even if that means unloading another object to make room. See "Automatic Loading and Unloading of Objects" on page 1-21 for more information.

Sharing and Multiple Object References

Object-based systems can use large amounts of memory, especially when an application needs to create and use many objects. To minimize redundancy and excess memory use, QuickDraw GX supports the sharing of objects.

For example, you may want to create a set of shape objects that are of different sizes and geometries, but that all have the same color and are drawn with the same transfer mode. You can create a single ink object with the desired color and transfer mode that all the shapes can reference, without having to create a separate ink object for each shape. In that situation there is one reference to the ink object for each shape that uses it.

Alternatively, your application can create data structures that contain object references, and two or more structures can contain references to the same object. For example, different palette structures can contain references to the same color set object that defines the palette colors. In that situation there is one reference to the color set object for each palette that uses it.

Sharing is not the same as making a copy. No matter how many references there are to an object, it is still only a single object. When you change any aspect of a shared object, those changes are reflected in every other object or data structure that references that object.

Object sharing provides at least three advantages:

- n It reduces memory use. Some objects that are used by many other objects are quite large. For example, a font, which can be a very large object, can be used by several different styles. And each style can be used by several text shapes.
- n It gives uniform behavior. For example, several shapes can share the same transform object, which causes each shape to be drawn in a specific relationship to each other, scaled and rotated in the same way, and so on.
- n It allows quick and efficient changes to the characteristics of multiple objects that share the same reference. For example, if several shape objects reference the same ink object, you need only change the color in the ink object to change the color of all shapes that reference the same ink.

Owner Count

The current number of references to an object is called its **owner count**. QuickDraw GX tracks and manages owner counts for you, so in most cases you needn't worry about how many references there are to an object and whether or not to delete it from memory when you no longer need it in a given context.

When you first create an object (with a call such as `GXNewStyle`), QuickDraw GX gives it an initial owner count of 1. Whenever you attach that object to another object (with a call such as `GXSetShapeStyle`), QuickDraw GX does not duplicate it; instead, it increases the object's owner count by 1. Whenever you delete that object (with a call such as `GXDisposeStyle`) or any object that references it (with a call such as `GXDisposeShape`), QuickDraw GX decreases its owner count by 1.

QuickDraw GX uses the owner count to determine when an object is no longer needed and can be deleted. If at any time the object's owner count decreases to zero, QuickDraw GX deletes it from QuickDraw GX memory. As far as your application is concerned, you create and dispose of objects as you wish, and let QuickDraw GX decide when to actually remove them from memory.

There can be cases, however, in which the owner count would normally become 0 but you do not want the object to be deleted. In those cases, you can increase owner count with the cloning capability of QuickDraw GX, described next.

Cloning

Although QuickDraw GX can correctly track owner counts as objects are created, disposed of, and referenced from other objects, it cannot know how many references to a given object exist in variables and data structures that you have created. In these situations, it is up to you to manage the owner counts of the objects that you use. Also, you may want to preserve a reference to an object that QuickDraw GX disposes of when it disposes of or modifies another object. In such a case, you can make sure the owner count of an object correctly reflects the number of references to it by **cloning** the object, which means increasing its owner count.

For example, if you create a color set object, it has an owner count of 1. If you dispose of that color set, its owner count becomes zero and it is deleted by QuickDraw GX, as it should be. On the other hand, if you assign a new ink object to a shape, that shape's original ink object is disposed of and the owner count of the new ink object is increased by 1. If you had wanted to maintain a reference to the shape's original ink object, you could have cloned that ink before assigning the new ink to the shape. The original ink's owner count would remain above zero, and it would therefore not be deleted.

As another example, you may temporarily change the style object assigned to a shape, intending to restore that style to the shape eventually. When you assign the new style object, QuickDraw GX decrements the original style object's owner count because it is no longer used by the shape. If the original style is not used by another object, its owner count would become 0 and QuickDraw GX would delete it. To prevent that from occurring, you can clone the original style object before assigning the new one.

QuickDraw GX cannot determine when you are finished with an object once it is cloned. If you clone an object, you are responsible for disposing of it when it is no longer needed.

Some Objects Cannot Be Cloned

Some objects have no owner count because they need to be able to be deleted even when valid references to them remain. View-related objects (view ports, view devices, and view groups) and fonts are examples of such shared objects that cannot be cloned. For example, suppose a transform object references a particular view port object associated with a window. When the application closes the window, it disposes of the view port. The view port object is deleted, even though a valid reference to it still remains in the transform object. (Subsequent drawing to that view port reference has no effect; QuickDraw GX ignores references to a view-related object that does not exist.) u

Automatic Loading and Unloading of Objects

Another way that QuickDraw GX minimizes memory requirements is by moving objects back and forth between memory and external storage as needed. If QuickDraw GX needs additional memory to create new objects, it can unload objects that are already in memory. When unloaded, an object is moved from computer memory to temporary private storage on disk. When loaded, that object is restored to normal object form in memory.

Typically, QuickDraw GX unloads objects that have not been accessed recently before unloading objects that your application has been using frequently. Also, for shape objects, QuickDraw GX provides flags that you can set to notify QuickDraw GX that you want it to unload a given shape before all others, or unload it after all others, when more memory is needed.

To reference and use an object, you needn't be concerned with or even know whether it is in a loaded or unloaded state. QuickDraw GX automatically loads any unloaded object when it is needed, even if that means unloading another object to make room.

For some purposes, such as measuring the storage size of an object, you may need to have the object in memory. Conversely, in other situations you may wish to allow an object to leave memory temporarily, to make more room in the QuickDraw GX heap. QuickDraw GX provides functions (such as `GXLoadShape` and `GXUnloadShape`) that you can use to explicitly load or unload an object.

The `GXLoadShape` and `GXUnloadShape` functions, and other loading and unloading calls, are described in the memory management chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*. The flags that affect the loading and unloading priority for shapes are described under shape attributes in the chapter “Shape Objects” in this book.

Direct Access to Object Structure: Locking and Unlocking

Normally, to modify a property of an object takes three steps. First, you make a function call to obtain a copy of the information in application memory. Then you modify the information. Finally, you make another function call to place that information back into the object in QuickDraw GX memory.

As a convenience, QuickDraw GX allows you to directly access parts of certain objects in QuickDraw GX memory in three specific situations: you can manipulate the geometric structure of a shape object, you can manipulate the profile data of a color profile object, and you can manipulate the contents of a tag object, without first having to work on copies of the data in application memory.

This direct manipulation is convenient, especially if you want to avoid copying large amounts of information, but it has a price. You must first lock the item you are accessing, so that it cannot be moved while you are working on it. When you have finished your alterations, you must be sure to unlock the item so that QuickDraw GX is free to relocate it. In the case of shape geometry, you must then make an additional call to QuickDraw GX to notify it that you have changed the shape.

Another drawback is that you cannot change the size of the item you are manipulating. If you need to make a shape’s geometry or a tag’s contents larger or smaller, you need to access the information in the normal way, through QuickDraw GX functions.

Remember also that locking an object fragments the QuickDraw GX heap, which can result in lower performance and possibly an error condition. Furthermore, in low-memory conditions, QuickDraw GX can actually unlock locked objects and move them if it needs to.

For information about locking shape objects, see the chapter “Shape Objects” in this book. For information about locking color profile objects, see the chapter “Colors and Color-Related Objects” in this book. For information about locking tag objects, see the chapter “Tag Objects” in this book.

External Storage of Objects: Flattening and Unflattening

QuickDraw GX objects exist (as objects) only in memory. You must convert a QuickDraw GX shape (a shape object and its referenced objects) into an equivalent compressed description in order to save it to external storage, transmit it across a network, or store it in the Clipboard. This process of converting objects to a compressed format that is no longer object-based is called **flattening**. The flattened information is a stream-based description with a public format, so that applications can share the data and reconstruct the objects from which the flattened stream was generated.

The data of flattened objects follows the format defined in the stream format chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*. To reconstruct a shape's object-based description from its flattened stream, you can manually create and initialize a set of objects based on the information in the stream, or—if QuickDraw GX is available—you can use QuickDraw GX functions to do it automatically.

Printing objects are also flattened and unflattened as the documents they are associated with are closed and reopened. For more information, see the core printing features chapter of *Inside Macintosh: QuickDraw GX Printing*.

Portable digital documents (PDDs) are specialized versions of print files, which are the flattened versions of documents sent to printers. For more information, see “Printing With QuickDraw GX” beginning on page 1-34, and the advanced printing features chapter of *Inside Macintosh: QuickDraw GX Printing*.

Fonts are represented in QuickDraw GX as font objects, which are flattened for transmission to printers or for external storage. A flattened font's format, however, is not related to the QuickDraw GX stream format. For more information, see the fonts chapter of *Inside Macintosh: QuickDraw GX Typography*.

Drawing and Hit-Testing Shapes

Ultimately, you need QuickDraw GX to draw the shapes that you create with it, and you may also need to respond to user manipulation of those drawn shapes. For that reason, QuickDraw GX provides several drawing functions and several kinds of hit-testing capabilities. This section summarizes the QuickDraw GX drawing process and the QuickDraw GX approach to hit-testing.

Drawing

Drawing is the process of converting the internal representation of a shape into an image on an output device. As noted in Figure 1-2 on page 1-12, a QuickDraw GX shape consists of several other objects in addition to a shape object. When you draw a shape, QuickDraw GX uses information from those objects and others to control how the shape is rendered. It uses the information in this order:

- n the geometry of the shape object
- n stylistic and color information from the style object and ink object
- n clipping and mapping information from the transform object
- n mapping and clipping information from one or more view port objects
- n mapping and clipping information from one or more view device objects

Drawing starts with geometry, a property of every shape object. The geometry defines the intrinsic dimensions of the shape. Those dimensions can then be modified, in several stages, until the rendered image appears on the screen or printer. The rest of this section describes in more detail how a shape's geometry is transformed as it passes through the drawing steps.

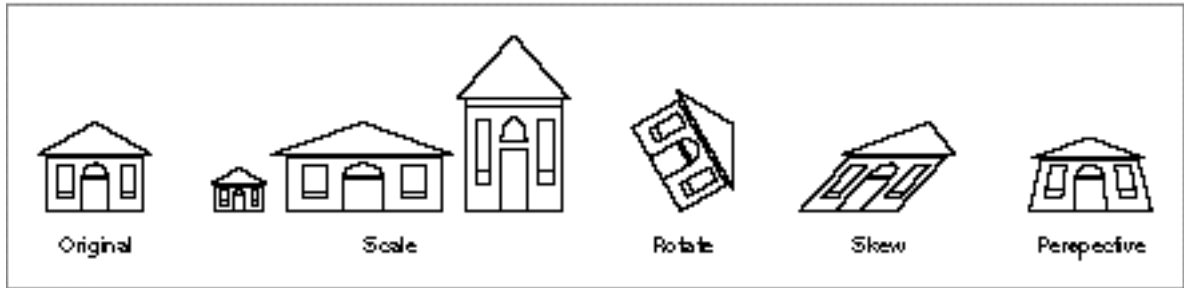
Mapping and Clipping

Mapping and clipping are two of the principal modifications a shape undergoes as it is prepared for drawing, and each occurs at several steps along the way.

A **mapping** is a 3×3 matrix that performs a mathematical transformation on a set of two-dimensional points, such as the geometry of a shape. Given any shape, you can use a mapping to control

- n translating, or moving, the shape from one (x, y) location to another
- n scaling the shape in the x-direction, y-direction, or both directions
- n rotating the shape around any point
- n skewing the shape
- n changing the perspective of the shape

Figure 1-4 shows examples of the effects of mapping.

Figure 1-4 Effects of mapping

The transform object, the view port object, and the view device object each has a mapping as a property. Each object's mapping can affect the location, orientation, scale, and other distortion of the shape as it evolves from geometry to rendered image (described under "The Drawing Sequence: Coordinate Conversion" beginning on page 1-28). Mappings are described more fully in the chapter "Transform Objects" in this book, and in the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

Clipping is the restriction of the visible part of a shape to a specific area. The **clip** is the specific description of that visible area. Clips are often rectangles or similar simple shapes, although QuickDraw GX permits clipping to any definable shape geometry (rectangle, polygon, path, and so on), which allows for very sophisticated clipping effects. Clips can even be glyph shapes and one-bit-per-pixel bitmaps. For the rules and restrictions on clips, see the chapter "Transform Objects" in this book.

The transform object, the view port object, and the view device object each has a clip as a property. Each object's clip is applied at a specific point during the preparation of the shape for drawing (described under "The Drawing Sequence: Coordinate Conversion" beginning on page 1-28). Each further restricts the part of the shape that will ultimately be visible.

View-Related Objects

The transform object associated with each QuickDraw GX shape contains a reference to one or more view port objects. When you draw the shape, QuickDraw GX uses that view port reference to determine at what position on which physical device or devices to draw the shape. To do that requires that the view port and two other view-related objects, the view group and view device, interact as follows:

- n A **view port** object represents a drawing environment. A view port is analogous to a porthole on a ship. The view port has a mapping that defines the scale, orientation, and location of the porthole, and a clip that prevents anything beyond the edges of the porthole from being drawn. If you think of a view port as analogous to a Macintosh graphics port, the view port mapping defines the location (in QuickDraw global coordinates) of the port on the screen, and the clip defines the visible region of the port. Unlike graphics ports, however, view ports are device independent, and their mappings control much more than location: they can also define the scaling, rotation, skewing, and other distortion of shapes drawn in the view port.
- n A **view device** object typically represents an actual, physical output device such as a monitor or printer. It, too, has a mapping and a clip that define its location and its visible (drawable) area. You can think of a view device as analogous to the Macintosh screen, in which case the mapping defines the location of the screen origin (and the size of the pixels too), and the clip defines the screen bounding rectangle. When a shape is drawn, it appears on a view device if the shape's view port intersects the view device. The object that controls the relative positions of view ports and view devices is the view group.
- n A **view group** object represents a coordinate plane that provides dimensions and relative positions for view ports and view devices. A view group's coordinates have a specific dimension (unit distance is 1 point, or 1/72 inch). For all view devices that represent actual physical devices, QuickDraw GX defines their locations in the onscreen view group's coordinate plane. Your application then defines the locations of view ports on that plane, and thus controls whether or not the view ports are visible on the view devices. A view group is equivalent to the QuickDraw coordinate plane (or to an offscreen graphics world) on the Macintosh, and view group coordinates are analogous to QuickDraw global coordinates. However, unlike with QuickDraw, QuickDraw GX global coordinates have a specific dimension and are device independent.

Figure 1-5 shows schematically how these objects interact as a shape is drawn. A shape geometry that defines a vase, a gray color defined in the ink object, a thick pen width defined in the style object, and a scaling in the transform object's mapping combine to make an elongated image of the vase. A portion of the vase appears on screen, where the clips of the view port and view device overlap.

Figure 1-5 How QuickDraw GX draws a shape

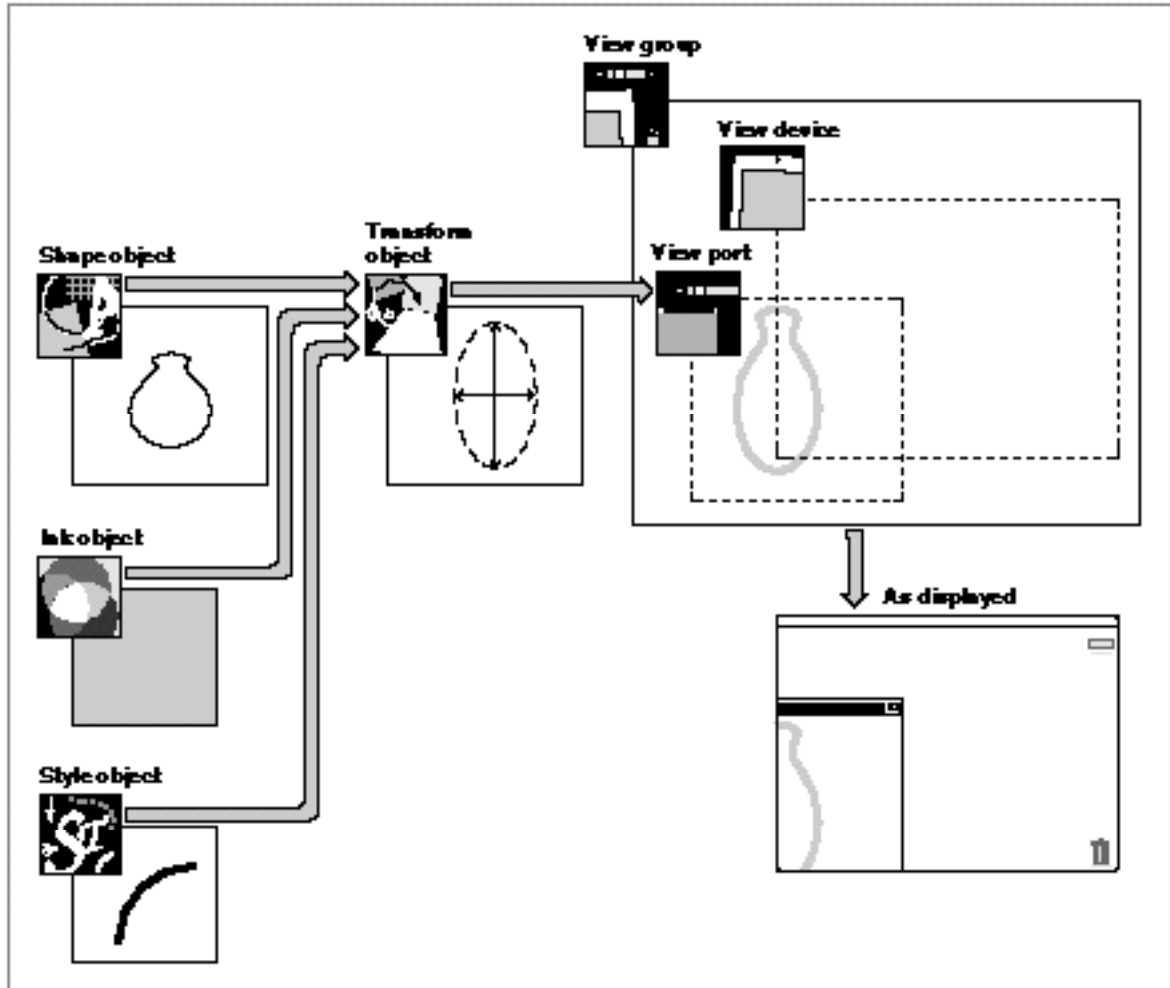


Figure 1-5 is a simple case in which a single shape and its transform are drawn to a single view port that partially intersects a single view device in the same view group. Quickdraw GX provides much greater flexibility, allowing for complex combinations of shapes, transforms, view ports, view devices, and even view groups:

- n Several shape objects can reference the same transform object, allowing these shapes to be scaled, rotated, and otherwise changed in unison.
- n Several transform objects can reference the same view port object, allowing shapes that are transformed in different ways to appear in the same view port.

- n A single transform object can reference several view port objects, allowing a single shape to appear simultaneously (even with different scaling or orientation) in several view ports.
- n View ports can exist in a hierarchy, in which one view port “contains” another, and thus its movement, scaling, and clipping affect view ports lower in the hierarchy.
- n Within a view group, view ports and view devices can overlap in any combination. Drawing occurs automatically wherever the visible portions of any view port and any view device overlap.
- n More than one view group can exist simultaneously, allowing for offscreen drawing. Furthermore, the view ports referenced by the transform of a single shape need not all be in the same view groups, allowing for simultaneous onscreen and offscreen drawing of a shape.

For further discussion and illustration of these display possibilities, see the chapter “View-Related Objects” in this book.

The Drawing Sequence: Coordinate Conversion

This section discusses the sequence of events, in terms of the mappings applied to a shape, that occur in drawing. To understand the details of the transformations that take place, you must understand the coordinate spaces whose relationships are determined by the mappings contained in various objects.

The information given in this section is an abbreviated version of the discussion of mapping and clipping in the chapter “View-Related Objects” in this book. Please see that chapter for additional information, especially about the role of clipping in drawing.

QuickDraw GX Coordinates

A **coordinate space** in QuickDraw GX consists of a plane in which positions are determined by coordinates. All coordinates in QuickDraw GX are specified with fixed-point numbers in the range of $-32,768.0$ to approximately $32,768.0$. Fixed-point numbers and the functions for manipulating them are described in the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*. Coordinates are always written in the order (x, y) , and for any coordinate space the point $(0.0, 0.0)$ represents the origin of the space. Points that lie to the right of the origin increase in a positive direction along the x-axis; points that lie below the origin increase in a positive direction along the y-axis.

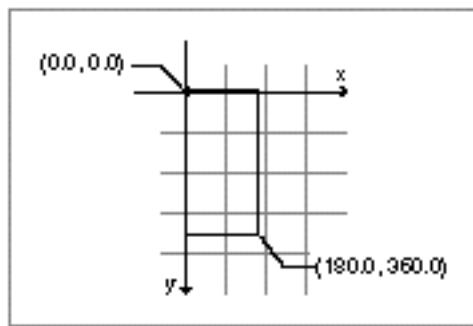
QuickDraw GX allows you to work in four coordinate spaces: geometry space, local space, global space, and device space. You can work separately in each space as appropriate; QuickDraw GX automatically converts among them when drawing. The spaces are described in order of their transformation during drawing.

Geometry Space

QuickDraw GX starts the drawing process by using the values in a shape's geometry. **Geometry space** is the space within which the fundamental position and dimensions of a shape object are defined. The numerical values in a shape's geometry define the shape's dimensions in geometry space.

Suppose, for example, that the geometry of a rectangle consists of the points $(0.0, 0.0)$ and $(180.0, 360.0)$, as shown in Figure 1-6. In geometry space, the rectangle's origin is at $(0.0, 0.0)$, its height is twice its width, and its area is 64,800.0 units square. No distance metric, such as points per inch, is defined for geometry space. Thus, the absolute size of a shape is undefined in geometry space.

Figure 1-6 A rectangle in geometry space

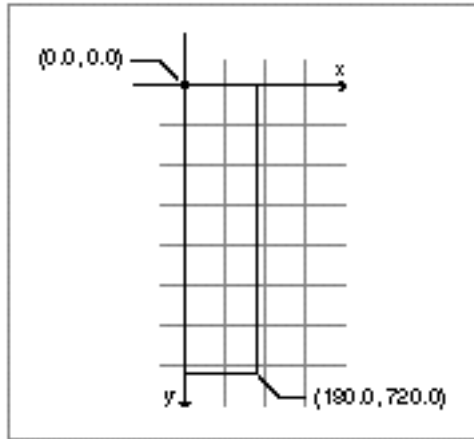


Geometry Space to Local Space

QuickDraw GX next modifies the shape's geometry by applying first the clip and then the mapping contained in the transform object attached to the shape. You typically use the transform's clip and mapping for application-specific purposes related to masking, moving, and distorting shapes within a document.

Local space defines the location and dimensions of a shape after it has been modified by the transform mapping (as well as the style properties and the transform clip). Because mappings can translate, scale, rotate, skew, and otherwise distort geometries, the dimensions of a shape in local space can be quite different from what they are in geometry space.

For example, if the rectangle shape discussed in the previous section had an associated transform whose mapping did nothing but scale the shape by 2.0 in the y-direction, its coordinates in local space would be $(0.0, 0.0)$ and $(180.0, 720.0)$, as shown in Figure 1-7. Its origin in local space would still be at $(0.0, 0.0)$, but its height would be four times its width, and its area would be 129,600.0 units square. Like geometry space, local space has no distance metric. The absolute size of a shape is still undefined in local space.

Figure 1-7 A rectangle in local space (transform mapping applied)

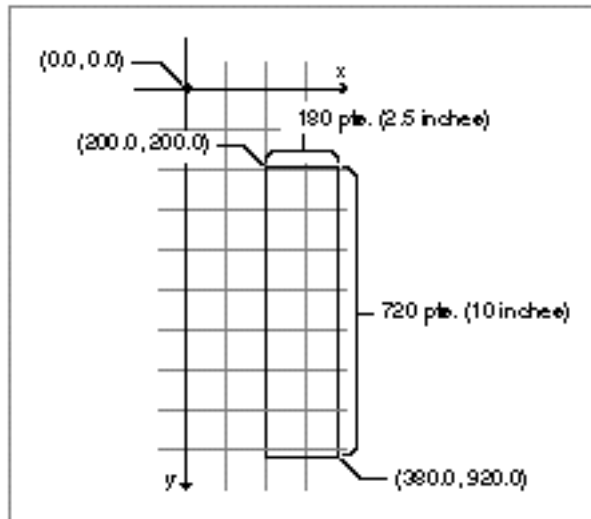
The transform object includes a reference to a view port object, and local space orients a shape within its view port. Local space is the coordinate system interior to, or local to, that view port—hence the name *local*. Thus, the rectangle example in this section would have the same local coordinates—that is, the same position and shape within its view port—no matter how the view port itself might be scaled or distorted by its own mapping when it is converted to global space (described next).

Local Space to Global Space

QuickDraw GX next modifies the shape's dimensions by applying first the mapping and then the clip contained in the view port object attached to the shape's transform. You typically use the view port's mapping to position the contents of the window you are drawing into, and you use its clip to restrict drawing to the interior of the window.

Global space defines the location and dimensions of a shape after the mapping (and clip) in its associated view port has been applied. Global space defines the real-world location and dimensions of a shape: coordinate values in global space represent distance in points (72 per inch) from the origin of the view group that the view port is part of. (Because it is the view group that relates view ports to view devices, objects in global space can have a specific spatial relationship with view devices, as described in the next section.)

For example, if the view port associated with the rectangle shape discussed in the previous sections had a mapping that did nothing but move the shape horizontally by 200.0 and vertically by 200.0, the shape's coordinates in global space would be (200.0, 200.0) and (380.0, 920.0), as shown in Figure 1-8. Its origin in global space would then be at (200.0 points, 200.0 points), its height would still be four times its width, and its area would be 129,600.0 points square (25 square inches).

Figure 1-8 A rectangle in global space (view port mapping applied)

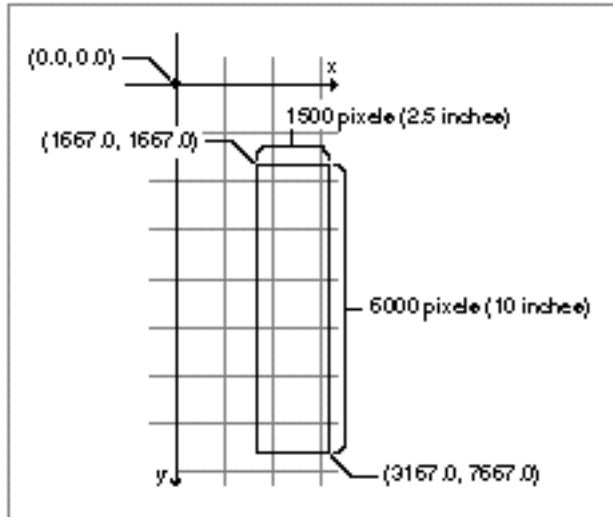
Thus, once a shape's dimensions have been converted from geometry space to local space to global space, they have a specific size and location and spatial relationship to other shapes in that view group. What remains for drawing, then, is for QuickDraw GX to convert this absolute (but device-independent) information to device-specific locations on output devices with specific pixel resolutions. That's where device space comes in.

Global Space to Device Space

Finally, QuickDraw GX modifies the shape's dimensions by applying first the mapping and then the clip of any view device object in the same view group as the view port. **Device space** defines the location and dimensions of a shape as displayed on a particular output device. The upper-left corner of the displayable area of a view device is at coordinate (0.0, 0.0) in device space. Unit distance between coordinates in device space represents one picture element, or pixel.

The view device's mapping defines both its location in global space (as a translation factor) and its pixel size (as a scaling factor). For example, if your device is a 600 dots-per-inch printer, QuickDraw GX converts global space to device space when drawing by scaling each pixel by 8.33333, which is $600/72$.

If the view device to which the rectangle shape discussed in the previous sections is drawn has a mapping that specifies no translation and a scale factor of 8.33333 both horizontally and vertically, that means that the view device's upper left corner is at (0.0, 0.0) in global space and its pixel resolution is 600 per inch. In device space, then, the dimensions of the rectangle would be (1667.0, 1667.0) and (3167.0, 7667.0), as shown in Figure 1-9.

Figure 1-9 A rectangle in device space (view device mapping applied)**Identity mapping**

A mapping that contains values such that it has no effect at all when applied to a shape is called the **identity mapping**. If the identity mapping is used for all mappings involved in drawing, a shape's geometry directly defines its absolute size and position (in points), and the shape is rendered on a view device at a resolution of 72 pixels per inch. ^u

It is seldom necessary to work in device space unless you are manipulating or hit-testing device bitmaps, because QuickDraw GX performs this kind of conversion for you. Most commonly, you define shapes in geometry space (using shape geometry), you position and modify them in local space (using the transform mapping), and you position and scale their view ports in global space (using the view port mapping).

Hit-Testing

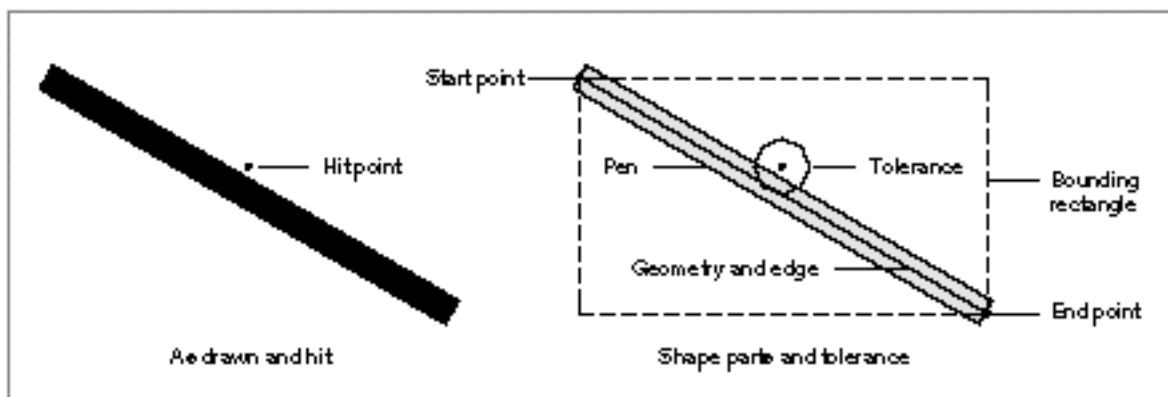
Hit-testing is the process of converting a point in the displayed representation of a shape to a location in the shape object's geometry. For example, when the user clicks the mouse button, hit-testing can tell you what displayed shape, and which part of that shape, the cursor was close to at the moment of clicking. You use hit-testing to select shapes or specific parts of shapes for highlighting or user manipulation, or to position the caret in text and to highlight text ranges. In a sense, hit-testing is the opposite of drawing, because it is a conversion from display representation to internal representation.

When you hit-test a shape, QuickDraw GX generally allows you to determine which part of a shape's geometry corresponds (within a certain tolerance) to the point you are testing against. **Tolerance** is the distance from a shape or shape part that a hit point can be and still be considered a successful hit. QuickDraw GX provides the following hit-testing functions:

- n `GXHitTestShape` tests a point in local space against a shape's geometry.
- n `GXHitTestPicture` tests a point in local space against a picture shape.
- n `GXHitTestLayout` tests a point in local space against the text of a layout shape.
Note that you can also use `GXHitTestShape` to test layout shapes, but the kind of information it returns is different from what `GXHitTestLayout` returns.
- n `GXHitTestDevice` tests a pixel (a point in device space) against a shape's geometry.

When you use a hit-testing function that returns a shape part, such as `GXHitTestShape`, the parts of a shape's geometry that you can hit-test for depend on the kind of shape. For example, for a typographic shape, the possible parts could be the bounding box, left side, right side, or side bearing of a glyph. For a line, the possible parts include its bounding rectangle, its geometry, its pen area, and its edges. Figure 1-10 shows the parts of a line involved in a particular hit-test. Shape parts are described in more detail in the chapter "Transform Objects" in this book.

Figure 1-10 Parts of a line for hit-testing



When you set up a hit-test using `GXHitTestShape`, you specify a tolerance and you also specify which parts of the shape to test against. The `GXHitTestShape` function returns all specified parts that are within the distance of the hit point defined by the tolerance. For example, if the hit point in Figure 1-10 is less than the tolerance away from the geometry part, the function could determine that the hit point corresponds to the bounds part, the geometry part, the pen part, and the edge part, depending on which of those shape parts you specify in the test.

The `GXHitTestShape` function analyzes shape parts in a specific order, and returns the distance from the hit point to the first part it encounters that is considered a hit. If you want to know the distance the hit point is from the pen, for example, you need to exclude both the bounds and the geometry parts from the test, because `GXHitTestShape` tests those first.

The `GXHitTestShape` function is described in the chapter “Shape Objects” in this book. The `GXHitTestPicture` function is described in the picture shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*. The `GXHitTestLayout` function is described in the layout carets, highlighting, and hit-testing chapter of *Inside Macintosh: QuickDraw GX Typography*. The `GXHitTestDevice` function is described in the chapter “View-Related Objects” in this book.

Printing With QuickDraw GX

From the point of view of your application, printing with QuickDraw GX is not fundamentally different from other types of drawing. The functions you use for drawing to the screen are the same functions you use for sending images to a printer. The printing component of QuickDraw GX allows you to draw shape objects and to use the information in other objects (such as style, ink, transform, and color set) in the same way you do when drawing to the screen. When printing, the printer is represented by view port and view device objects, just as in other drawing.

To control these printing capabilities, your application creates printing-related QuickDraw GX objects before it prints a document for the first time. Your application flattens and stores those objects when it saves the document, and it retrieves and unflattens those objects when it reopens the document. The objects include the job object (the primary holder of printing information), the format object (specifying scaling and page dimensions), and the paper-type object (specifying a paper-type name and dimensions). These objects also include references to **collection objects**, which are similar to QuickDraw GX objects but are managed by the Collection Manager. The Collection Manager is described in the Collection Manager chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

QuickDraw GX prepares an document for printing by **spooling** it, which means flattening its shapes and storing them along with the associated printing objects as a **print file**. To actually print the document, QuickDraw GX **despools** the print file and sends its data to the printer. QuickDraw GX can also use the printing process create a **portable digital document (PDD)**, which is a kind of print file that contains sufficient object and font information that it can be displayed or printed on any QuickDraw GX system, regardless of what fonts or printers are installed.

QuickDraw GX printing is based on a message-passing architecture. For example, QuickDraw GX sends a message when it wants to print a page, display a dialog box on the user’s screen, or initialize a job object. Therefore, in addition to manipulating printing objects and collection objects, your application needs to be able to respond to QuickDraw GX messages for some basic printing actions, such as updating windows behind dialog boxes.

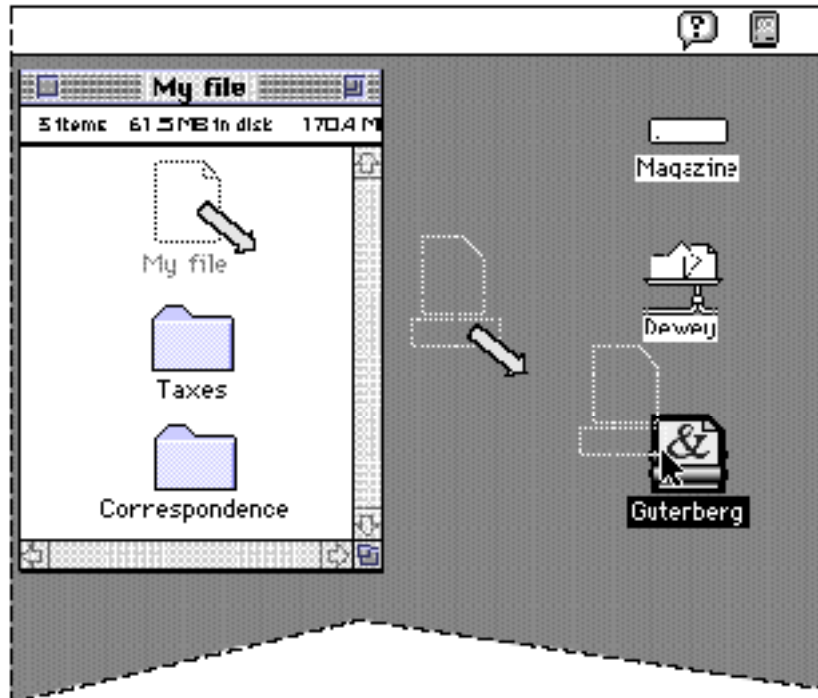
Printing extensions and printer drivers also use printing messages. A **printing extension** is an add-on software module that allows you to extend the printing functionality provided by applications and printer drivers. A **printer driver** controls how the contents of a document are spooled, rendered, and sent to a specific output device. The messaging technology used with QuickDraw GX is described in the Message Manager chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*. How printing extensions and printer drivers use printing messages, and information on how to write an extension or driver, are described in *Inside Macintosh: QuickDraw GX Printing Extensions and Drivers*.

The rest of this section summarizes the QuickDraw GX printing features of most interest to application developers. For more information on printing than is provided here, see *Inside Macintosh: QuickDraw GX Printing*.

Core Printing Features

QuickDraw GX provides several core features you can use to implement basic printing capabilities:

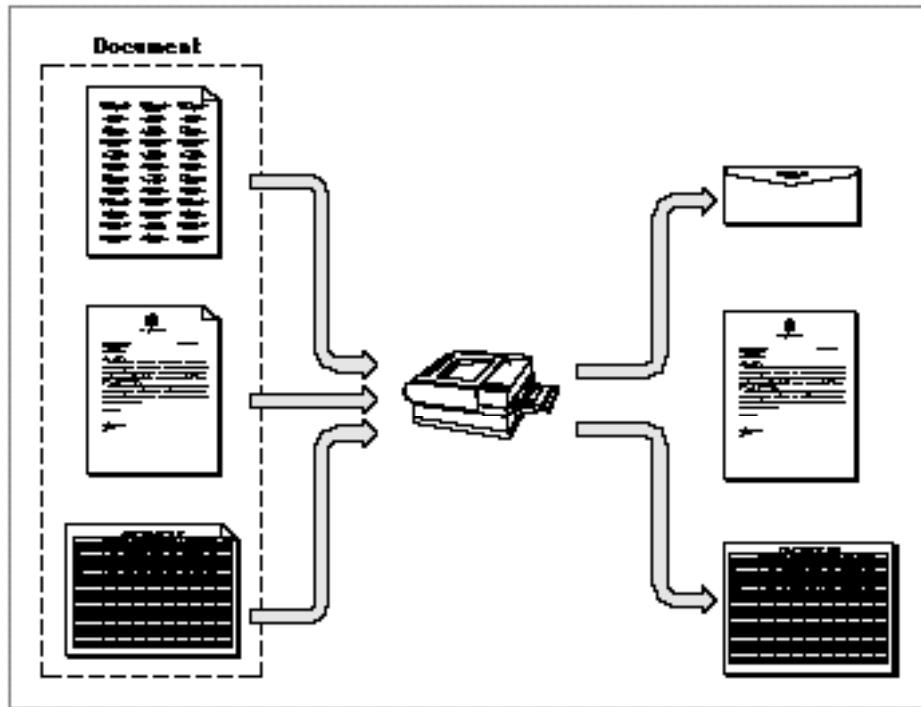
- n You can create and manipulate common QuickDraw GX printing objects. For example, you create a job object for every printable document, and that job object references two printer objects: a formatting printer and an output printer. QuickDraw GX allows a user to specify a formatting printer that is different from the output printer, so that QuickDraw GX can consistently format to the device used for final output, while permitting drafts to be printed on a different printer.
- n You can print documents in either of two ways. If your application stores each page as a single picture shape, you can print a page at a time with a single command. Otherwise, you can print each page by drawing, in turn, all the shapes that make up that page. QuickDraw GX captures those drawing commands and sends the images to the printer.
- n You can display QuickDraw GX printing dialog boxes. You use QuickDraw GX functions to display these expandable, movable modal dialog boxes that allow users to view windows that would otherwise be obscured, and you override a QuickDraw GX printing message to permit updating of windows behind the dialog box as it is moved. For general information on movable modal dialog boxes, see the Dialog Manager chapter of *Inside Macintosh: Macintosh Toolbox Essentials*.
- n You can support printing to desktop printers. A **desktop printer** is represented by an icon on the user's desktop. To print a document to a desktop printer, a user drags a document to the desktop printer icon, or else selects it and chooses the Print command from the Finder's File menu. A user can create multiple desktop printers. Figure 1-11 shows the document "My File" being printed to the desktop printer "Gutenberg."

Figure 1-11 Dragging a document to a desktop printer icon on the desktop

Custom Dialog Boxes and Page Formats

QuickDraw GX allows you to customize some of its printing features to address the needs of your particular application:

- n You can add panels to QuickDraw GX dialog boxes, to provide special features that require additional user specification. For example, your application can add a panel that provides special color options for the user to select, such as color separation and color choices.
- n You can manipulate the objects that handle page formatting, allowing users to specify unique formats for individual pages of a printable document. For example, your application can allow a user to create and print a single document that consists of an address page on an envelope, a business letter on a page in portrait orientation, and a spreadsheet on a page in landscape orientation. See Figure 1-12 for an example of this.

Figure 1-12 Printing a single document that has multiple formats


Advanced Printing Features

QuickDraw GX includes several advanced printing features that allow your application to provide additional capabilities for users and to optimize output for particular printers:

- n You can use direct mode printing, which takes advantage of a printer's built-in features, such as fast text streaming with built-in fonts.
- n You can use alternative representations of QuickDraw GX objects. When printing, QuickDraw GX translates the objects of a shape into device-specific information. For optimum performance on particular devices, you can assign specialized tag objects known as **synonyms** to printed shapes and associated objects, to provide an alternative representation of the graphics objects. You can also tag objects to select specific printing options, such as pen table information, for vector devices.
- n You can display your own printing status information. QuickDraw GX allows you to prevent the display of the standard QuickDraw GX Status dialog box during printing and to substitute status information from your own application.
- n You can open and display the pages of a PDD or other print file. If QuickDraw GX is installed, any application—including applications that are not QuickDraw GX-aware—can create a document that can be viewed or printed from any other computer that has QuickDraw GX installed. The PDD also provides font security in that the font data is “locked” into the document and only the minimum font information is contained therein.

The QuickDraw GX Programming Environment

QuickDraw GX is more than a framework for creating and manipulating objects; it is also a programming environment with many features designed to aid application development. This section describes some of these features and some ways to approach programming with QuickDraw GX.

Setting Up QuickDraw GX Memory

Your application enters the QuickDraw GX environment by creating a graphics client. A **graphics client** is an object that represents a memory environment set up for your application by QuickDraw GX. It consists of a QuickDraw GX heap and the global variables needed by QuickDraw GX. It represents your application's individual QuickDraw GX world.

Normally, each application creates and uses a single graphics client, although it is possible to create and use more than one at a time. In most cases, you don't even explicitly set up a graphics client at all; one is created for you as you begin making QuickDraw GX calls to create and use objects. For more information, see the memory management chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

Handling Errors

In all but its printing component, QuickDraw GX uses a sophisticated, three-level system for reporting diagnostic messages. The execution of a function may result in the generation of an **error**, a **warning**, or a **notice**:

- n Errors represent the most severe problems, and occur when a function is unable to execute.
- n Warnings occur when a function has completed but may have provided an incorrect or unexpected result.
- n Notices occur when unnecessary or redundant actions have been performed. (Notices are available only in the debugging version of QuickDraw GX; see "Debugging and Non-Debugging Versions" on page 1-39 for more information.)

Errors, warnings, and notices are not returned as function results. Instead, they are **posted**, or stored by QuickDraw GX in locations accessible through function calls. To determine whether, for example, an error has occurred, your application makes a specific call (such as `GXGetGraphicsError`) that returns not only the most recent error but also the first error posted since the last time you called `GXGetGraphicsError`. For information about these function calls, see the debugging chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

You can use the error-handling facilities of QuickDraw GX in the following ways:

- n You can install an error-handling function that QuickDraw GX calls whenever an error, warning, or notice occurs.
- n You can post errors, warnings, or notices yourself from your own application's functions.
- n You can tell QuickDraw GX to ignore specific warnings or notices. You can create and manipulate a list of warnings and a list of notices to be ignored.

Functions within the printing component of QuickDraw GX do not use this system for reporting diagnostics. Instead, most functions place errors directly into the job object involved. Some printing functions return a function result of type `OSErr`, that describes a Macintosh error code. For more information, see the core printing features chapter of *Inside Macintosh: QuickDraw GX Printing*.

Debugging

QuickDraw GX provides both a debugging and non-debugging version of the software. In addition, QuickDraw GX provides a low-level debugger, similar to MacsBug, that allows you to examine internal data structures. This section summarizes these approaches to debugging. For more information, see the debugging chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

Debugging and Non-Debugging Versions

There are two versions of QuickDraw GX. The debugging version is intended for application development and is meant for use by software developers only. The non-debugging version is intended for running completed applications and is the publicly released version of QuickDraw GX.

The debugging version of QuickDraw GX provides extensive error handling. It posts all three levels of diagnostic messages (errors, warnings, and notices), and it provides special functions to assist in the posting, utilization, and control of debugging messages. The debugging version allows you to perform validation checking on both QuickDraw GX objects and your own application parameters at each function call. The debugging version also includes the `GXGetShapeDrawError` function, which can give you very specific information on why a particular shape may not have drawn correctly.

The non-debugging version of QuickDraw GX has much less extensive error handling. It reports only two levels of result messages (errors and warnings), and only a limited number of them. In the non-debugging version, errors and warnings are mostly limited to out-of-memory and range-checking messages.

Debugging With GraphicsBug

GraphicsBug is a tool you can use to track down bugs in a QuickDraw GX application. Its mode of use and its command set are analogous to MacsBug. GraphicsBug works with both the debugging and non-debugging versions of QuickDraw GX.

You can use GraphicsBug to check the contents of QuickDraw GX memory and to display and validate objects within memory. GraphicsBug does not allow you to create, modify, or dispose of objects. Listing 1-1 shows a sample dump of the QuickDraw GX heap created with GraphicsBug.

Listing 1-1 Sample GraphicsBug heap dump (HD) listing

Start	Length	Typ	Busy	Mstr	Ptr	Temp	TBsy	Disk	Object
00469728	0000010c+00	d		00000000			b		heap header block
00469834	0000003c+00	d		00000000					freeFileList
00469870	0000005c+00	i		00470e68					text
004698cc	00000042+02	i		00470e64					text
00469910	000000a0+00	i		00470e60					style
004699b0	00000036+02	i		00470e5c					ink
004699e8	00000060+00	i		00470e58					transform
00469a48	000000c0+00	d		00000000					port
00469b08	00000038+00	i		00470e54					full
00469b40	00007228	f		00000000					free block
00470d68	00000110+00	d		00469728			b		master pointer block
00470e78	0000000c+00	d		00469728			b		heap trailer block

	Total	Blocks	Total	of	Block	Sizes
Free	0001	#	1	00007228	#	29224
Direct	0002	#	2	00000318	#	792
Indirect	0006	#	6	00000210	#	528
Sub Heaps	0000	#	0	00000000	#	0
Heap Size	0009	#	9	0000775c	#	30556

The listing shows the objects that you create as well as private QuickDraw GX objects. From the heap dump, you can look into the contents of these objects using additional GraphicsBug commands. For a complete list of commands, type ? on the GraphicsBug command line.

Note

Do not use GraphicsBug to make assumptions about the structure of objects in memory; object structure is subject to change. u

For examples of the use of GraphicsBug in analyzing flattened shapes, see the stream format chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

Programming Conventions and Consistencies

The QuickDraw GX programming environment provides many consistent features and conventions to make graphics software development more convenient and more efficient. This section lists some of them.

Object Behavior

Many QuickDraw GX objects have similar features and consistent behavior, in ways such as the following:

- n In general, setting a property of an object causes action or new behavior only when needed—which may not be immediately. For example, setting the `gxDiskShape` attribute of a shape object, which instructs QuickDraw GX to write the shape to disk, takes effect only as soon as QuickDraw GX needs memory space and looks for objects to unload.
- n QuickDraw GX handles object properties consistently; it does not change a property once you have set it. For example, if you set an object reference to `nil` in order to use the default version of an object, QuickDraw GX does not replace that `nil` value with an actual reference. If you later make another call to retrieve that reference, you will get back the `nil` value you originally set. (A minor exception to this rule occurs with certain calls that assign arrays to the style object associated with typographic shapes; QuickDraw GX may alter the order of elements in those arrays. See the layout styles chapter in *Inside Macintosh: QuickDraw GX Typography* for more information.)
- n Most objects can be explicitly moved into and out of memory, using `GXLoadObject` and `GXUnloadObject` functions. View-related objects and printing objects are exceptions to this rule.
- n Many objects can have tag objects attached to them, using `GXGetObjectTags` and `GXSetObjectTags` functions. View group objects, printing objects, font objects, and tag objects themselves are exceptions to this rule.
- n Most objects can be shared, and thus have `GXCloneObject` and `GXGetObjectOwners` functions. View-related objects are exceptions to this rule; they are shared but they have no owner count and cannot be cloned.

Functions and Function Results

QuickDraw GX functions are designed to operate in a consistent manner, as follows:

- n Most QuickDraw GX functions do not return error codes as function results. Instead, they return object references or pointers to structures. This makes nesting of calls easier.
- n Functions are consistently named and have consistent behavior across all objects. Most objects have similarly behaving `GXNewObject`, `GXDisposeObject`, `GXCopyToObject`, and `GXEqualObject` functions. The property-accessing `GXGetObjectProperty` and `GXSetObjectProperty` functions behave consistently, using index values and ranges for inserting, deleting, or replacing all or parts of arrays.
- n All functions, except some printing functions that return an `OSErr` value, return `nil` or zero as a function result if an error occurs.

- n If a function posts an error, it does not modify any data or objects that are input parameters to the function.
- n Functions of the form `GXGetObjectProperty` that fill out an array that is passed as a parameter typically return the number of elements in that array as a function result. For example, the `GXGetTransformViewPorts` function, used to get the list of view port references in a transform object, returns the number of elements in the list as its function result. Thus, you commonly call such a function twice in a row: first to determine the size of array to allocate, and second to obtain the filled-out array itself.
- n Many functions of the form `GXSetObjectProperty`, which modify a property of a particular object, have a parallel function of the form `GXSetShapeProperty`, which performs the same function but allows you to specify instead the shape object that references the affected object. If the object whose property is changed is not shared by more than one shape, the two functions have an identical effect. If, however the object is shared, `GXSetShapeProperty` makes a copy of the object before modifying it, so that the other shapes using it are not affected unintentionally.
 For example, the `GXSetTransformViewPorts` function assigns a list of view port references to the specified transform object, and the `GXSetShapeViewPorts` function assigns a list of view port references to the transform object associated with the specified shape. If the transform object is used by more than one shape, `GXSetTransformViewPorts` has the effect of altering all shapes that use that transform; `GXSetShapeViewPorts`, however, first makes a copy of the transform and then alters it, so that other shapes are not affected.
- n When an out-of-memory condition occurs, it is rarely fatal. QuickDraw GX initially posts an `out_of_memory` error, but execution continues and subsequent attempts to reference the object responsible for the error result in posting of `object_is_nil` errors.
- n The debugging version of QuickDraw GX provides extensive error-checking and validation capabilities to help you determine why a function call has failed.

Function Parameters

When passing parameters to a QuickDraw GX function, you can take advantage of the following design consistencies and conveniences:

- n The first parameter in any function call is the object or structure acted upon.
- n Parameters whose names contain the word “source” are never modified by a function; parameters whose names contain the word “target” may be modified.
- n Whenever an array or structure is passed as a parameter to a function, the application is responsible for allocating it. QuickDraw GX fills out structures, but it does not allocate them.
- n When a variable-sized array or structure is passed as a parameter to a function, it is preceded in the parameter list by a size or count parameter.

- n In the C language definitions for QuickDraw GX functions, the term `const` preceding a parameter that is an array, structure, or pointer indicates that QuickDraw GX reads from, but does not write to, the data pointed to by the parameter.
- n In general, passing `nil` or zero for a parameter instructs QuickDraw GX to use its default or most appropriate behavior for that situation. Thus you need to explicitly set parameters only when you need specific, non-default behavior. To actually assign a `nil` value to a property, pass the constant `gxSetToNil` (see Table 1-1).
- n In functions that use coordinates, the x-coordinate (horizontal axis) is specified before the y-coordinate (vertical axis).
- n For convenience in handling arrays and pointers in parameters, QuickDraw GX provides several predefined constants, as listed in Table 1-1.

Table 1-1 Convenience constants for parameters

Constant	Value	Explanation
<code>gxSelectToEnd</code>	<code>-1</code>	Used in a size or count parameter, to mean “from the current position in the array to the end of the array.”
<code>gxSetToNil</code>	<code>(void *)(-1)</code>	Used in a parameter (where a function takes more than one parameter) to assign a <code>nil</code> value to a pointer or reference property (simply passing <code>nil</code> has no effect on the property).
<code>gxNoAttributes</code>	<code>0</code>	Used to clear the attributes property of an object.
<code>gxColorValue1</code>	<code>0xFFFF</code>	Used to specify the maximum value for a color-component, which is interpreted to mean 1.0
<code>gxAnyNumber</code>	<code>1</code>	Used as an index in an array declaration to indicate that the array is not of any specific size.

Implementation limits

Limits on valid parameter values or on the sizes of structures or behaviors of objects may depend on the current implementation of QuickDraw GX, and may be different from the fundamental limits imposed by the programmatic interface itself. For example, a parameter to a function may be a long, but the range of acceptable values for that parameter may be much smaller than the full range of values that can fit into a long. u

Code Naming Conventions

QuickDraw GX uses these naming conventions to provide consistency across the application interface:

- n Function names begin with uppercase GX—for example, `GXDrawShape`. Important exceptions are those in the Collection Manager and those that are mathematical functions because those functions can be useful outside of the QuickDraw GX environment.
- n Identifiers of constants and data types defined by QuickDraw GX begin with lowercase gx—for example, `gxWindingFill` and `gxShapeType`. One exception is the type `Fixed`, which represents a QuickDraw GX fixed-point number but does not have a gx prefix. Types defined by the programming language itself, such as `short`, do not have a gx prefix.
- n Names of fields in data structures, and parameter names in function prototypes, begin with lowercase letters and do not have a gx prefix.
- n An enumeration that defines several constants is usually named with a plural form—for example, `gxDashAttributes`. Such an enumeration is commonly paired with a type definition that is a singular form of the same name—for example, `gxDashAttribute`. You can use the type to specify one of the enumerated values for a parameter or field.
- n Object attributes have suffixes that identify the kind of object they apply to. For example, dash attributes specified by the `gxDashAttributes` enumeration include the attributes `gxBendDash` and `gxAutoAdvanceDash`.

Relationship to the Macintosh Toolbox

QuickDraw GX is in general designed to be platform independent. Within the QuickDraw GX environment, the programming interface does not depend on the existence of the Macintosh Toolbox or Macintosh hardware.

However, when running on a Macintosh computer, QuickDraw GX still must have an interface with the Macintosh Toolbox. QuickDraw GX does not create windows, handle menus, receive keystrokes or automatically track mouse movements (although it does support hit-testing). Therefore, for basic input and output needs, QuickDraw GX includes several sets of functions that carry information from the QuickDraw GX environment to the Macintosh world and back:

- n You can associate QuickDraw GX view ports with Macintosh windows, which restricts a view port to the current size of the window and prevents drawing outside the content area of the window. QuickDraw GX and the Macintosh Window Manager then manage the view port for you such that, if the user moves the window, the view port moves too, or if the user changes the size of a window, the drawable area in the view port also changes.
- n You can associate a QuickDraw GX view device with a Macintosh graphics device (`GDevice`).

Introduction to QuickDraw GX

- n You can translate coordinate locations, including mouse locations, between the integer-based QuickDraw global space and the fixed-point QuickDraw GX coordinate spaces.
- n You can convert QuickDraw calls to QuickDraw GX calls, in two ways. You can use one set of functions to set up a situation whereby all QuickDraw calls are captured and converted to QuickDraw GX shapes in a QuickDraw GX picture. You can use another function to directly translate QuickDraw pictures to QuickDraw GX pictures.

See the Macintosh environment chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities* for information about these functions.

Summary Table and Diagram of QuickDraw GX Objects

QuickDraw GX provides at least 17 objects that you can manipulate. Table 1-2 lists these objects and summarizes their characteristics. Following Table 1-2, Figure 1-13 on page 1-49 diagrams the relationships among the basic QuickDraw GX objects, and shows the object properties of each.

Table 1-2 QuickDraw GX objects

Object	Description
Basic QuickDraw GX objects	
Shape	Defines the basic representation of a drawable entity. A shape object describes a geometry of a certain type (such as a line, rectangle, bitmap, or text) and how the geometry is framed or filled when drawn. A shape also has references to its three related objects: style, ink, and transform. See the chapter “Shape Objects” of this book for more information. Graphic shape types are described in <i>Inside Macintosh: QuickDraw GX Graphics</i> ; typographic shape types are described in <i>Inside Macintosh: QuickDraw GX Typography</i> .
Style	Describes certain characteristics affecting how a shape is drawn. For geometric shapes, this includes the thickness of the pen, the starting and ending caps for line segments, joins between line segments, and the dash or pattern to be applied to the shape. For typographic shapes, it includes the font, text size, and typeface of the text. See the chapter “Style Objects” in this book, the geometric styles chapter of <i>Inside Macintosh: QuickDraw GX Graphics</i> , and the typographic styles and layout styles chapters of <i>Inside Macintosh: QuickDraw GX Typography</i> for more information.

continued

Table 1-2 QuickDraw GX objects (continued)

Object	Description
Ink	Describes a shape's color and its transfer mode (how the color is applied when the shape is drawn). Ink objects support many different kinds of color specification, and many different transfer modes. An ink object can reference a color set object or color profile object or both. See the chapters "Ink Objects" and "Color-Related Objects" in this book for more information.
Transform	Describes the clip and mapping applied to a shape when it is drawn. The clip limits the extent of the shape when it is drawn; it may be described by any primitive shape geometry (except picture, text, layout, and multi-bit bitmap). The mapping defines translation, scaling, skewing, rotation or perspective. The transform object also describes the criteria used for hit-testing the shape. Transforms have references to one or more view port objects. See the chapter "Transform Objects" in this book for more information.
Color set	Contains an indexed set of colors; analogous to a color table. Color sets are used when colors are specified by index instead of by direct color value. Bitmaps commonly use color sets. See the chapter "Colors and Color-Related Objects" in this book for more information.
Color profile	Contains color matching information. The information in a color profile can be used to convert device-specific colors to device-independent colors and back. To provide the most faithful reproduction of colors on different devices, QuickDraw GX automatically performs color matching with available color profiles whenever it draws. See the chapter "Colors and Color-Related Objects" in this book for more information.
View port	Defines the location into which a shape is drawn. A view port object describes the clip and mapping associated with a window (or a part of a window, such as a pane). The mapping defines the location, scale, and orientation of the view port in QuickDraw GX global coordinates. A view port specifies the dithering or halftones used by every object that draws into this window. View ports can be arranged in a hierarchy. See the chapter "View-Related Objects" in this book for more information.
View device	Describes the clip, mapping, and bitmap associated with a physical display device such as a monitor or printer. The mapping describes the view device's position and resolution in QuickDraw GX global coordinates. The bitmap defines the dimensions of the device, the number of bits per pixel, the color representation of each pixel value, and the color profile. See the chapter "View-Related Objects" in this book for more information.

Table 1-2 QuickDraw GX objects (continued)

Object	Description
View group	Describes an imaging world that is the global space in which view ports and view devices are located. Within a view group, view ports and view devices can overlap each other in any combination; the intersection of each view port with a view device determines what is actually visible on that device. Multiple view groups allow for offscreen drawing, in which view ports or view devices can have the same positions without interfering with each other, since they are in different coordinate spaces. See the chapter “View-Related Objects” in this book for more information.
Tag	Contains any kind of information an application wants to add to a QuickDraw GX object. Tag objects are general containers that can have anything in them, from labels to alternate drawing instructions to anything else you feel is useful. You can attach a tag object to the tag list of most kinds of objects (except tag objects themselves). See the chapter “Tag Objects” in this book for more information.
Printing objects	
Job	Holds the primary printing information for a document. Every printable document has a job object associated with it. The job object specifies a number of copies and a page range, and includes references to one or more format objects and two printer objects. See the core printing features chapter of <i>Inside Macintosh: QuickDraw GX Printing</i> for more information.
Format	Specifies page-formatting characteristics such as scaling and page dimensions, and includes a reference to a paper-type object. See the core printing features chapter of <i>Inside Macintosh: QuickDraw GX Printing</i> for more information.
Paper type	Specifies a paper-type name (such as “US Letter”), the physical dimensions of the paper, and the printable area within it. See the core printing features chapter of <i>Inside Macintosh: QuickDraw GX Printing</i> for more information.
Printer	Represents the capabilities of a physical printer and includes a name and type, a driver name and type, and a reference to one or more view device objects that represent imaging areas, and from which you can retrieve information. See the advanced printing features chapter of <i>Inside Macintosh: QuickDraw GX Printing</i> for more information.
Print file	Represents the file that results from spooling, which is the preparation of a printable representation of a document. See the advanced printing features chapter of <i>Inside Macintosh: QuickDraw GX Printing</i> for more information.

continued

Table 1-2 QuickDraw GX objects (continued)

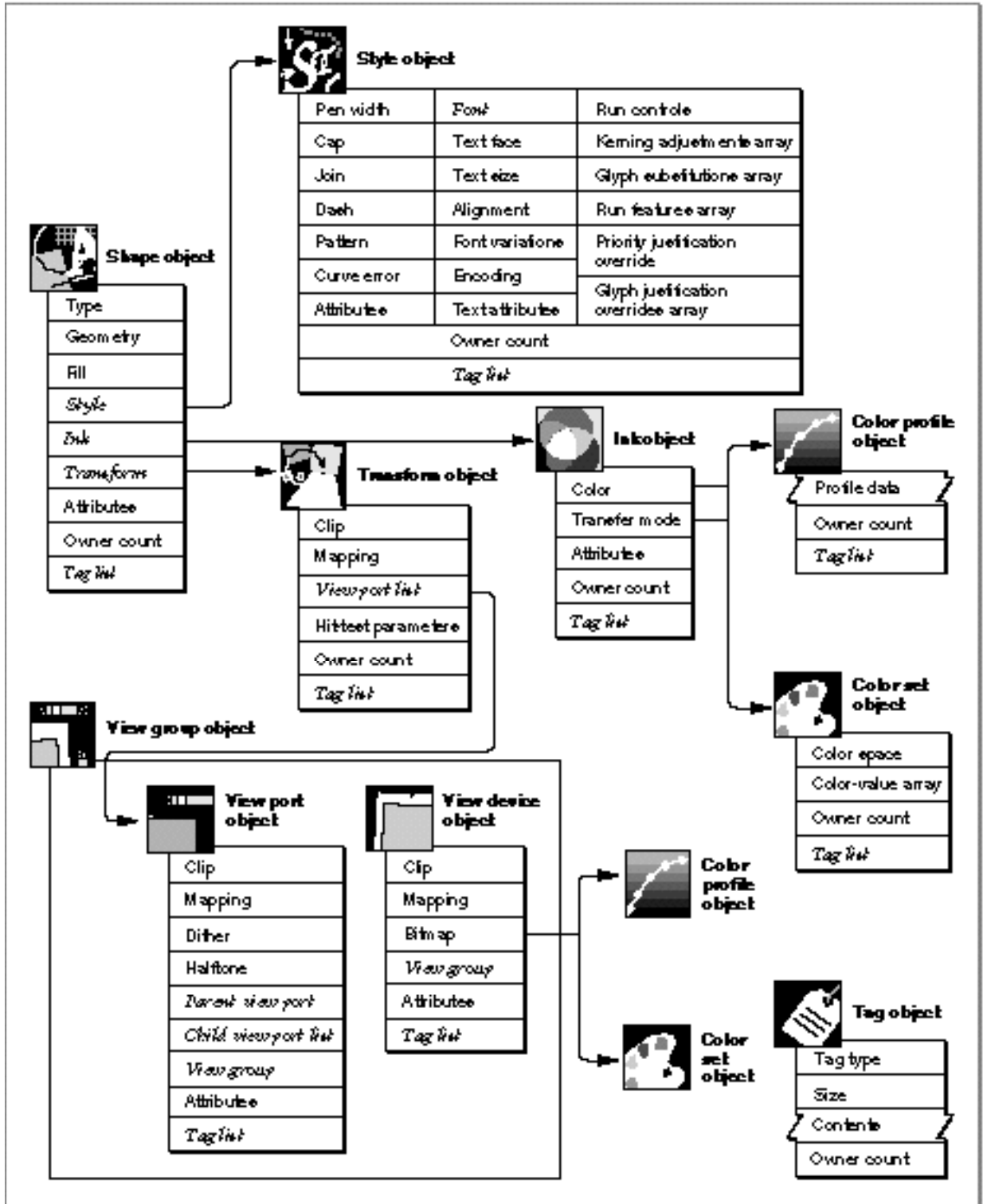
Object	Description
Other objects	
Font	Represents an available font. A font object contains information about the font's names, encodings, font variations, and other tables. See the fonts chapter of <i>Inside Macintosh: QuickDraw GX Typography</i> for more information.
Graphics client	Represents the QuickDraw GX memory allocated for an application, which is separate from the memory the application itself occupies and allocates. Each QuickDraw GX application is represented by a graphics client object. A graphics client has no accessible properties. See the memory management chapter of <i>Inside Macintosh: QuickDraw GX Environment and Utilities</i> for more information.
Collection	Contains any type of data in any structure. Used by printing objects to hold additional information such as halftoning specifications. Collection objects are not QuickDraw GX objects; they are managed by the Collection Manager. See the Collection Manager chapter of <i>Inside Macintosh: QuickDraw GX Environment and Utilities</i> for more information.

The following figure, Figure 1-13, shows the relationships among the basic QuickDraw GX objects and lists the properties of each object. The appropriate portion of this figure is reproduced in each chapter that describes a specific kind of object.

Note that, in Figure 1-13, properties that are references (or arrays of references) to other objects are shown in italics; for most of those properties, an arrow extends to the diagram of the referenced object. For clarity, however, some of the arrows are not shown. For example, no object's tag list has an arrow pointing to the diagram of the tag object. For the same reason, the properties of the view port that reference other view ports have no attached arrows.

For a diagram showing all the properties of the printing objects, see the introductory chapter of *Inside Macintosh: QuickDraw GX Printing*. For a diagram showing the contents of the geometry of each type of shape object, see the chapter "Shape Objects" in this book.

Figure 1-13 Properties of the basic QuickDraw GX objects



Shape Objects

Contents

About QuickDraw GX Shapes	2-5
About Shape Objects	2-7
Shape Properties	2-7
Shape Type	2-9
Shape Geometry	2-11
Shape Fill	2-13
Shape Attributes	2-16
Default Shapes	2-18
Modifying Shape Properties	2-19
Drawing Shapes	2-20
Hit-Testing Shapes	2-20
Saving and Restoring Shapes	2-22
Using Shape Objects	2-22
Creating and Manipulating Shape Objects	2-22
Getting and Setting the Default Shape Objects	2-23
Creating and Disposing of Shape Objects	2-24
Getting the Size of a Shape Object in Memory	2-25
Copying, Comparing, and Cloning Shape Objects	2-25
Caching Shape Objects	2-27
Loading and Unloading Shape Objects	2-27
Manipulating Shape Object Properties	2-28
Getting and Setting a Shape Object's Type, Fill, and Attributes	2-28
Copying the Geometry From One Shape to Another	2-29
Getting and Setting a Shape Object's Style, Ink, and Transform	2-30
Resetting a Shape Object's Properties to Their Default Values	2-31
Manipulating a Shape Object's Owner Count	2-31
Getting and Setting a Shape Object's Tag References	2-32
Converting Shapes From One Type to Another	2-32
Directly Manipulating a Shape's Geometry	2-34

Drawing and Hit-Testing Shapes	2-35
Drawing Shapes	2-35
Hit-Testing Shapes	2-36
Flattening and Unflattening Shapes	2-39
Shape-Related Functions Described Elsewhere	2-42
Shape Objects Reference	2-45
Constants and Data Types	2-45
The Shape Object	2-46
Shape Type	2-46
Shape Fill	2-46
Shape Attributes	2-47
Flatten Flags	2-48
The Spool Block	2-49
The Hit-Test Info Structure	2-50
Functions	2-51
Creating and Manipulating Shape Objects	2-52
GXGetDefaultShape	2-52
GXSetDefaultShape	2-53
GXNewShape	2-54
GXDisposeShape	2-55
GXGetShapeSize	2-56
GXCopyToShape	2-57
GXCopyDeepToShape	2-58
GXEqualShape	2-60
GXCloneShape	2-61
GXCacheShape	2-62
GXDisposeShapeCache	2-63
GXGetShapeCacheSize	2-64
Manipulating Shape Object Properties	2-65
GXGetShapeType	2-66
GXSetShapeType	2-66
GXSetShapeGeometry	2-67
GXGetShapeFill	2-68
GXSetShapeFill	2-69
GXGetShapeStyle	2-69
GXSetShapeStyle	2-70
GXGetShapeInk	2-71
GXSetShapeInk	2-71
GXGetShapeTransform	2-72
GXSetShapeTransform	2-73
GXGetShapeAttributes	2-74
GXSetShapeAttributes	2-74
GXResetShape	2-75
GXGetShapeOwners	2-76
GXGetShapeTags	2-77
GXSetShapeTags	2-78

CHAPTER 2

Directly Manipulating a Shape's Geometry	2-80
GXLockShape	2-80
GXUnlockShape	2-81
GXGetShapeStructure	2-82
GXChangedShape	2-83
Drawing and Hit-Testing Shapes	2-84
GXDrawShape	2-84
GXHitTestShape	2-86
Flattening and Unflattening Shape Objects	2-87
GXFlattenShape	2-88
GXUnflattenShape	2-90
Application-Defined Spool Function	2-91
MySpoolProc	2-91
Summary of Shape Objects	2-93
Constants and Data Types	2-93
Functions	2-95
Application-Defined Spool Function	2-97

This chapter describes shape objects and the functions you can use to manipulate them. Read this chapter if you create or use any kind of QuickDraw GX shapes.

Before reading this chapter, you should be familiar with the information in the chapter “Introduction to QuickDraw GX” in this book. For more information on graphic shapes, see *Inside Macintosh: QuickDraw GX Graphics*. For more information on typographic shapes see *Inside Macintosh: QuickDraw GX Typography*. Additional information relevant to the storage of shape objects is in the stream format chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

This chapter introduces the concept of a QuickDraw GX shape, and describes shape objects and their properties. It then shows how to use general QuickDraw GX shape-manipulation functions to

- n create and manipulate shape objects
- n manipulate shape object properties
- n directly manipulate shape geometry
- n draw and hit-test shapes

This chapter also lists and cross-references all shape-related QuickDraw GX functions that are described elsewhere in this book and in other parts of *Inside Macintosh*.

About QuickDraw GX Shapes

Shapes are fundamental to the QuickDraw GX object architecture. To draw or print in QuickDraw GX requires creating and manipulating QuickDraw GX shapes. A **shape** is a drawable graphic or typographic entity that you create with QuickDraw GX objects.

Shapes come in two general categories: graphic and typographic. Graphic shapes are further subdivided into geometric shapes (points, lines, rectangles, polygons, and so on), bitmap shapes, and picture shapes. Typographic shapes are subdivided into text shapes, glyph shapes, and layout shapes. Table 2-1 on page 2-9 describes all the types of shapes recognized by QuickDraw GX.

This chapter discusses only shapes in general. The QuickDraw GX object architecture allows you to perform many operations on a shape without regard for what type of shape it is; those are the operations described here.

In the QuickDraw GX architecture, every shape includes four objects:

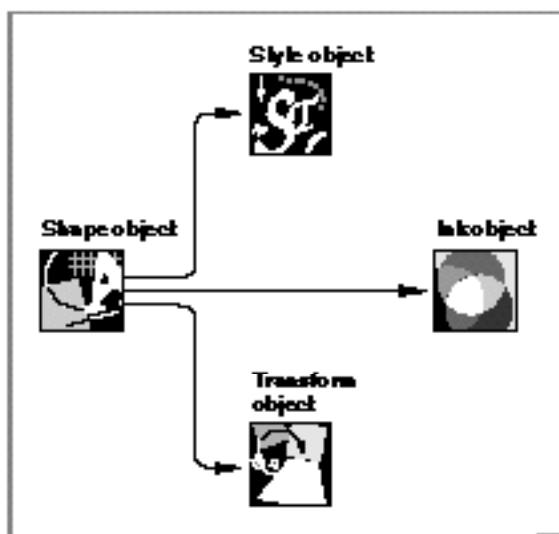
- n **Shape object.** A shape object describes the geometric structure or text content of a shape and contains references to the other objects that make up the shape.
- n **Style object.** A style object defines much of the appearance of a shape, such as the size of the pen with which it is drawn or the size of its text. See the chapter “Style Objects” in this book for more information.

Shape Objects

- n **Ink object.** An ink object defines the color and transfer mode to use when drawing a shape. See the chapter “Ink Objects” in this book for more information.
- n **Transform object.** A transform object defines how the appearance of a shape is altered (such as by clipping, scaling, or rotation) when it is drawn, and how the shape responds to mouse clicks. A transform object also contains references to the view port objects that describe where the shape is drawn. See the chapter “Transform Objects” in this book for more information.

Figure 2-1 shows the four objects used to represent a QuickDraw GX shape.

Figure 2-1 Basic components of a QuickDraw GX shape



The interface to each of these object types is entirely procedural—you cannot in most cases access any information in the objects directly. You must manipulate the items of information in an object, called the **properties** of the object, using QuickDraw GX functions.

The rest of this chapter describes the data types and functions you can use to create and manipulate shape objects and their properties.

Terminology Note

A QuickDraw GX *shape* is considered to be the combination of four objects just described. A *shape object* is one of the objects that make up the shape; it defines, among other characteristics, the shape’s *geometry*, which is the description of the specific dimensions and location of the kind of shape (line, curve, rectangle, and so on) that is to be drawn. u

About Shape Objects

This section describes the contents of the shape object and summarizes some of the main tasks you can perform with shapes.

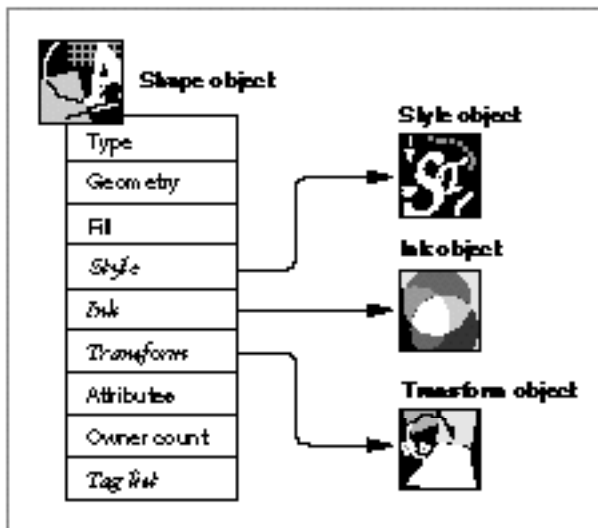
QuickDraw GX identifies an individual shape object through a shape **reference**. To obtain information about a shape object, you must send its reference as a parameter to a QuickDraw GX function (except that you can determine if two references identify the same shape object simply by comparing them for equality, and you can examine a reference to see if it is `nil`).

Shapes are device-independent. Their location, resolution, color, and other properties are not constrained by the characteristics of the display device to which they are drawn.

Shape Properties

The properties of a shape object for the most part define the basic geometric characteristics of the shape. Shape objects have nine accessible properties, as shown in Figure 2-2. Note that, because a shape is an object and not a data structure, the order of the properties as shown in Figure 2-2 is completely arbitrary. Properties in italics are references to other objects.

Figure 2-2 The shape object and its properties



Shape Objects

The first six properties are specific to shape objects alone. They determine a shape's geometric type, geometric structure, fill, and references to other objects:

- n **Type.** A value that specifies what type of geometry the shape object has. The different shape types include point, line, rectangle, text, glyphs, and so on. The section “Shape Type” beginning on page 2-9 describes the different shape types, and “Getting and Setting a Shape Object's Type, Fill, and Attributes” beginning on page 2-28 discusses how to manipulate this property.
- n **Geometry.** A set of values that describes the specific graphic structure of the shape. For example, the geometry of a point shape specifies the two coordinates of the point. The geometry of a text shape specifies the sequence of characters or glyphs that it contains. *Inside Macintosh: QuickDraw GX Graphics* discusses the geometries of graphic shapes and *Inside Macintosh: QuickDraw GX Typography* discusses the geometries of typographic shapes. See also Figure 2-3 on page 2-12 of this chapter for a summary of shape geometries. The geometry property differs from other properties in one important respect: you can edit it directly. See “Directly Manipulating a Shape's Geometry” beginning on page 2-34 for more details.
- n **Fill.** A value that determines how a shape is filled or framed when drawn. QuickDraw GX provides a number of different ways of filling a shape. For example, a rectangle shape might have a **solid fill**, which indicates that the shape represents a solid rectangle—that is, the entire area enclosed by the sides of the rectangle is included in the shape. Alternatively, a rectangle shape might have a **framed fill**, which indicates that the shape represents a hollow rectangle—only the lines connecting the rectangle's corners are included in this shape. The section “Shape Fill” beginning on page 2-13 discusses types of shape fills, and the section “Getting and Setting a Shape Object's Type, Fill, and Attributes” beginning on page 2-28 discusses how to manipulate the shape fill property of a shape object.
- n **Style, ink, and transform object references.** References to the style object, ink object, and transform object that are needed to complete the specification of the shape. The section “Getting and Setting a Shape Object's Style, Ink, and Transform” beginning on page 2-30 discusses how to manipulate these references.

The remaining three shape properties are common to many QuickDraw GX objects (including styles, inks, and others):

- n **Attributes.** A group of flags that control certain aspects of the behavior of the object. For a shape object, these flags allow you to specify where QuickDraw GX stores the shape object and how editing operations affect the shape object. For example, the `gxMemoryShape` attribute specifies that QuickDraw GX should avoid writing the shape object out to storage, and the `gxMapTransformShape` attribute indicates that certain editing operations, such as the `GXMoveShape` function, are to affect the data in the shape's transform object rather than the data in the shape itself. The section “Shape Attributes” beginning on page 2-16 describes the shape attribute flags, and the section “Getting and Setting a Shape Object's Type, Fill, and Attributes” beginning on page 2-28 discusses how to manipulate the attributes property of a shape object.

Shape Objects

- n **Owner count.** A number that indicates how many references to the object exist. The chapter “Introduction to QuickDraw GX” in this book includes general information about owner counts, and “Manipulating a Shape Object’s Owner Count” beginning on page 2-31 describes when and how to examine and alter a shape object’s owner count.
- n **Tag list.** A list of references to custom information about the object, stored in private data structures called **tag objects**. The chapter “Tag Objects” in this book describes tag objects in general and how you can use them to add custom information to objects. The section “Getting and Setting a Shape Object’s Tag References” beginning on page 2-32 discusses how to manipulate the tag objects associated with a shape object.

Shape Type

A shape object’s type property specifies what sort of geometry the shape has. Table 2-1 lists the defined constants for each shape type and describes what each one means. (Note that the empty and full shape types are unusual, in that they have no specific geometry; they are used for specialized operations other than drawing.) The constants are defined in the `gxShapeTypes` enumeration.

Table 2-1 Shape types

Constant	Value	Explanation
<code>gxEmptyType</code>	1	An empty shape. It has no geometry, no contents, and no bounds. The intersection of two shapes that do not touch is the empty shape. You can use empty shapes as a starting point for collecting graphic elements. This shape type is described in the geometric shapes chapter of <i>Inside Macintosh: QuickDraw GX Graphics</i> .
<code>gxPointType</code>	2	A point shape. Its geometry contains two fixed-point coordinate values representing the location of the point. This shape type is described in the geometric shapes chapter of <i>Inside Macintosh: QuickDraw GX Graphics</i> .
<code>gxLineType</code>	3	A line shape. Its geometry contains two point geometries—the starting point and the ending point. This shape type is described in the geometric shapes chapter of <i>Inside Macintosh: QuickDraw GX Graphics</i> .
<code>gxCurveType</code>	4	A curve shape. Its geometry contains three point geometries—a starting point, an ending point, and a control point—which together describe a quadratic Bézier curve. This shape type is described in the geometric shapes chapter of <i>Inside Macintosh: QuickDraw GX Graphics</i> .

continued

Table 2-1 Shape types (continued)

Constant	Value	Explanation
gxRectangleType	5	A rectangle shape. Its geometry contains four fixed-point values—representing the coordinates of the left, top, right, and bottom corners of the rectangle. This shape type is described in the geometric shapes chapter of <i>Inside Macintosh: QuickDraw GX Graphics</i> .
gxPolygonType	6	A polygon shape. Its geometry includes any number of separate multiple-point polygon contours, each contour consisting of straight line segments connecting its points. This shape type is described in the geometric shapes chapter of <i>Inside Macintosh: QuickDraw GX Graphics</i> .
gxPathType	7	A path shape. Its geometry includes any number of separate multiple-point path contours, each contour consisting of straight or curved line segments connecting its points. This shape type is described in the geometric shapes chapter of <i>Inside Macintosh: QuickDraw GX Graphics</i> .
gxBitmapType	8	A bitmap shape. Its geometry contains information about the bitmap's size, shape, and color, as well as the pixel image itself. This shape type is described in the bitmap shapes chapter of <i>Inside Macintosh: QuickDraw GX Graphics</i> .
gxTextType	9	A text shape. Its geometry contains a string of characters to be drawn with uniform stylistic properties such as font family, font size, and style. This shape type is described in the text shapes chapter of <i>Inside Macintosh: QuickDraw GX Typography</i> .
gxGlyphType	10	A glyph shape. Its geometry contains a string of glyphs, each of which may have its own typestyle when drawn. This shape type is described in the glyph shapes chapter of <i>Inside Macintosh: QuickDraw GX Typography</i> .
gxLayoutType	11	A layout shape. Its geometry contains a string of characters plus sophisticated formatting information that affects how the text is displayed. This shape type is described in the layout shapes chapter of <i>Inside Macintosh: QuickDraw GX Typography</i> .

Table 2-1 Shape types (continued)

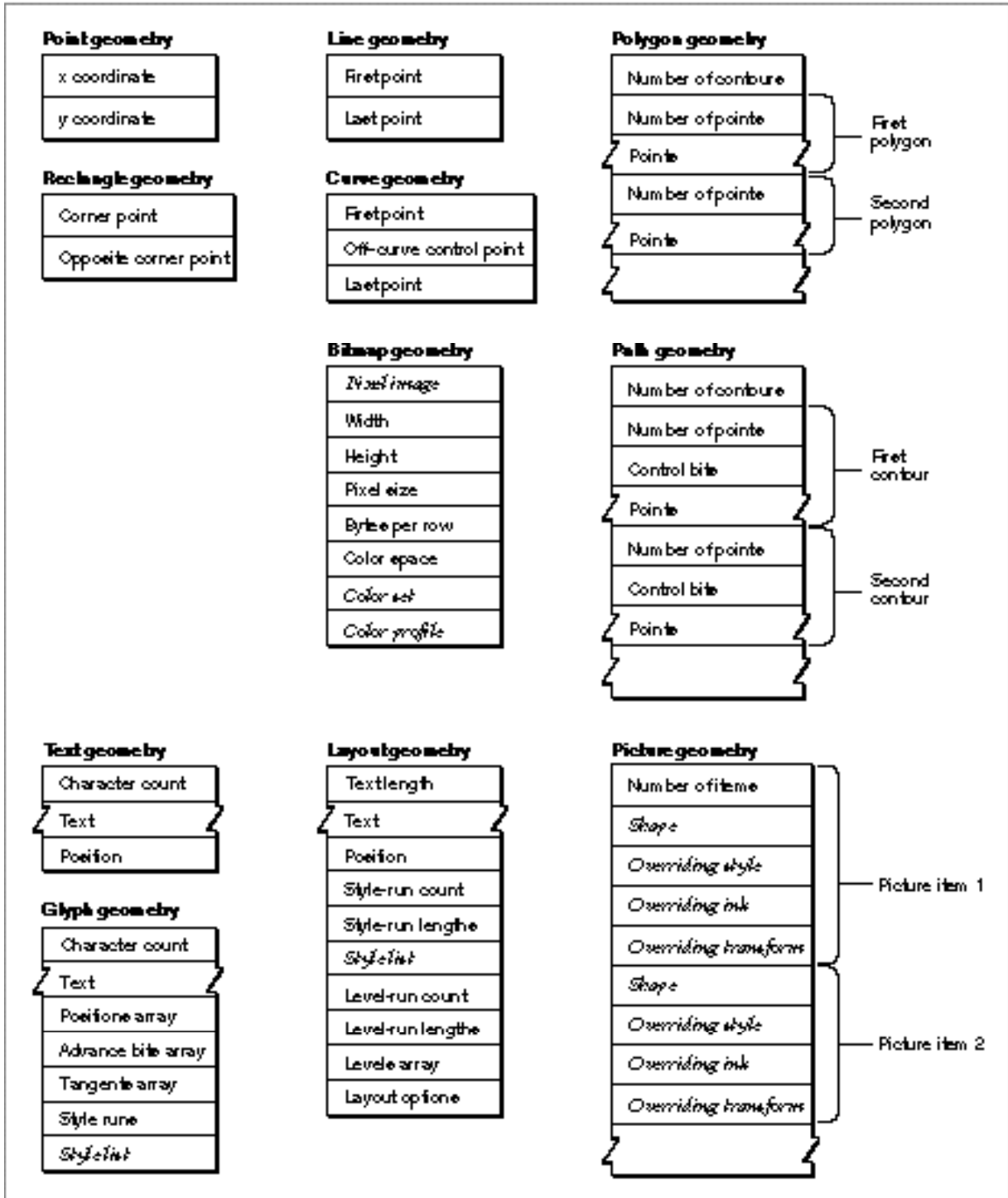
Constant	Value	Explanation
<code>gxFullType</code>	12	A full shape. It encompasses all of the QuickDraw GX coordinate space. Inverting an empty shape produces a full shape. The full shape contains every other shape and no other shape contains the full shape. You can use full shapes to specify the largest possible clip area for maximum visibility when drawing. This shape type is described in the geometric shapes chapter of <i>Inside Macintosh: QuickDraw GX Graphics</i> .
<code>gxPictureType</code>	13	A picture shape. It is a collection of other shapes, including possibly other picture shapes. This shape type is described in the picture shapes chapter of <i>Inside Macintosh: QuickDraw GX Graphics</i> .

Shape Geometry

Most shape geometries are defined in terms of points in a coordinate space. QuickDraw GX coordinates are pairs of fixed-point numbers (of type `Fixed`), as defined in the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*. QuickDraw GX coordinate spaces are described in the chapter “View-Related Objects” in this book.

Figure 2-3 summarizes the contents of the geometry property for each type of QuickDraw GX shape (except empty and full types, which have no geometry). In the figure, elements of the geometry that are references (or arrays of references) to other objects are shown in italics. For a diagram showing all the properties of the basic QuickDraw GX objects, see the chapter “Introduction to Objects” in this book. For a diagram showing all the properties of the printing objects, see the introductory chapter of *Inside Macintosh: QuickDraw GX Printing*.

Figure 2-3 Shape geometry for each type of QuickDraw GX shape



Shape Objects

The structures of individual shape geometries are specific to each shape type and thus are not described here. See the chapters of *Inside Macintosh: QuickDraw GX Graphics* and *Inside Macintosh: QuickDraw GX Typography* specified in Table 2-1 of the previous section for more information.

Nevertheless, some of the functions that affect shape geometry are common to all types of shapes, and are therefore described in this chapter. The section “Copying the Geometry From One Shape to Another” beginning on page 2-29 discusses how to copy the geometry between shapes. The section “Directly Manipulating a Shape’s Geometry” beginning on page 2-34 discusses how to get direct access to a shape’s geometry—as a data structure rather than as an object property—in order to modify it. Also, Figure 2-3 on page 2-12 of this chapter summarizes the contents of the geometry of each type of QuickDraw GX shape.

Shape Fill

Each shape object has a fill property. The shape fill specifies how QuickDraw GX interprets the geometry of the shape: how the shape is drawn, how the shape is hit-tested, and how certain geometric operations, like intersection or union, interpret the shape. Table 2-2 lists the defined constants for shape fill and describes what each one means. (Note that some shape fills have two or more equivalent names.) The constants are defined in the `gxShapeFills` enumeration.

Table 2-2 Shape fills

Constant	Value	Explanation
<code>gxNoFill</code>	0	No fill—the shape is not filled at all. QuickDraw GX does not draw a shape with this shape fill and you may not perform geometric operations on it. You can use this shape fill to temporarily hide shapes or to prevent parts of a picture from drawing.
<code>gxOpenFrameFill</code>	1	Open-frame fill—the shape is outlined instead of filled. With this shape fill, QuickDraw GX interprets the shape as a connected series of straight or curved lines from the starting point of the shape geometry to the ending point (but not back to the starting point again).
<code>gxFrameFill</code>	1	Framed fill (same as <code>gxOpenFrameFill</code>).
<code>gxClosedFrameFill</code>	2	Closed-frame fill—the shape is outlined instead of filled. As with the open-frame fill, QuickDraw GX interprets the shape as a series of lines (or curves) from the starting point of the shape geometry to the ending point. However, in this case, QuickDraw GX also includes a line (or curve) from the ending point to the starting point, thus closing the contour.

continued

Table 2-2 Shape fills (continued)

Constant	Value	Explanation
<code>gxHollowFill</code>	2	Hollow fill (same as <code>gxClosedFrameFill</code>).
<code>gxEvenOddFill</code>	3	Even-odd fill—the shape is filled using the even-odd rule. See Figure 2-4 for an illustration of this rule; see <i>Inside Macintosh: QuickDraw GX Graphics</i> for further explanation.
<code>gxSolidFill</code>	3	Solid fill (same as <code>gxEvenOddFill</code>).
<code>gxWindingFill</code>	4	Winding fill—the shape is filled using the winding-number rule. See Figure 2-4 on page 2-14 for an illustration of this rule; see <i>Inside Macintosh: QuickDraw GX Graphics</i> for further explanation.
<code>gxInverseEvenOddFill</code>	5	Inverse even-odd fill—the shape is filled in an opposite manner from the even-odd rule; everything <i>not</i> filled using the even-odd rule is filled using this rule. See <i>Inside Macintosh: QuickDraw GX Graphics</i> for further explanation.
<code>gxInverseSolidFill</code>	5	Inverse solid fill (same as <code>gxInverseEvenOddFill</code>).
<code>gxInverseFill</code>	5	Inverse fill (same as <code>gxInverseEvenOddFill</code>).
<code>gxInverseWindingFill</code>	6	Inverse winding fill—the shape is filled using the winding-number rule and then inverted. See <i>Inside Macintosh: QuickDraw GX Graphics</i> for further explanation.

Figure 2-4 Even-odd and winding fills

Shape Objects

Note that framed fill, hollow fill, and solid fill are alternative names for open-frame fill, closed-frame fill, and even-odd fill, respectively, and that both inverse solid fill and inverse fill are alternate names for inverse even-odd fill.

Not all shape fills make sense for all shape types. Table 2-3 shows the acceptable shape fills for each shape type (the alternative names are not listed). Note that empty and full shapes are permitted to have certain fill types even though they have no geometry.

Table 2-3 Valid shape fills for each shape type

Shape types	Valid shape fills
Empty	gxNoFill gxInverseEvenOddFill gxInverseWindingFill
Full	gxNoFill gxEvenOddFill gxWindingFill gxInverseEvenOddFill gxInverseWindingFill
Point, line, curve	gxNoFill gxOpenFrameFill
Rectangle	gxNoFill gxClosedFrameFill gxEvenOddFill gxWindingFill gxInverseEvenOddFill gxInverseWindingFill
Polygon, path	any shape fill
Text, glyph, layout	gxNoFill gxEvenOddFill gxWindingFill
Bitmap, picture	gxNoFill gxEvenOddFill

For additional examples of how different shape fills can affect the appearance of different types and geometries of shapes, see the geometric shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*.

Shape Attributes

Each shape object includes a property that is a set of attributes, a group of flags that specify certain aspects of the shape's behavior. Table 2-4 lists the defined shape attribute constants and describes what each one means. The constants are defined in the `gxShapeAttributes` enumeration.

Table 2-4 Shape attributes

Constant	Value	Explanation
<code>gxNoAttributes</code>	0x0000	No shape attributes are set. You can use this attribute to clear or test against the current value of a shape's attributes.
<code>gxDirectShape</code>	0x0001	QuickDraw GX is to load the shape into directly accessible memory. Set this flag for shape objects that you don't want stored in accelerator card memory, or whose geometric structures you want to manipulate directly (see "Directly Manipulating a Shape's Geometry" beginning on page 2-34). The attributes <code>gxDirectShape</code> and <code>gxRemoteShape</code> are exclusive; do not set them both.
<code>gxRemoteShape</code>	0x0002	QuickDraw GX is to load the shape into remote memory (memory used by an accelerator card), if possible. When this flag is set, the shape might draw faster but you might not be able to edit the shape's geometry directly (see "Directly Manipulating a Shape's Geometry" beginning on page 2-34). The attributes <code>gxRemoteShape</code> and <code>gxDirectShape</code> are exclusive; do not set them both.
<code>gxCachedShape</code>	0x0004	QuickDraw GX is to prepare the shape for the fastest possible drawing by caching it compressed in an offscreen bitmap. (See "Caching Shape Objects" beginning on page 2-27; also, compare this with using the <code>GXCacheShape</code> function, described on page 2-62.)

Table 2-4 Shape attributes (continued)

Constant	Value	Explanation
gxLockedShape	0x0008	QuickDraw GX is to prohibit changes to the shape's geometry or the shape's disposal. You can use this flag in the debugging version of QuickDraw GX to prevent accidental modification of a shape intended to be used as a constant. When this flag is set, you cannot use the geometry-editing functions described in the geometric shapes chapter of <i>Inside Macintosh: QuickDraw GX Graphics</i> and the text, glyph, and layout shapes chapters of <i>Inside Macintosh: QuickDraw GX Typography</i> . However, you can still alter the shape's geometric structure by accessing it directly (see "Directly Manipulating a Shape's Geometry" beginning on page 2-34).
gxGroupShape	0x0010	QuickDraw GX is to group all shapes within this shape as a single shape when hit-testing. This attribute applies to picture shapes only; for more information see the picture shapes chapter of <i>Inside Macintosh: QuickDraw GX Graphics</i> .
gxMapTransformShape	0x0020	QuickDraw GX is to apply shape-transforming operations to the shape's transform object rather than to the shape's geometry. This attribute is set by default for bitmap shapes, picture shapes, and layout shapes. See the chapter "Transform Objects" in this book for more information on applying transformations to shapes.
gxUniqueItemsShape	0x0040	QuickDraw GX is to create a complete copy of each shape added to this picture rather than simply creating a reference to the added shape. This attribute applies to picture shapes only; for more information see the picture shapes chapter of <i>Inside Macintosh: QuickDraw GX Graphics</i> .
gxIgnorePlatformShape	0x0080	QuickDraw GX is to treat the codes in the geometry of this shape as glyph codes rather than character codes. This attribute applies to typographic shapes only; for more information see the typographic shapes chapter of <i>Inside Macintosh: QuickDraw GX Typography</i> .

continued

Table 2-4 Shape attributes (continued)

Constant	Value	Explanation
<code>gxNoMetricsGridShape</code>	<code>0x0100</code>	QuickDraw GX is not to use hints (special display instructions) provided with the font used for this shape. Set this attribute if you intend to manipulate text as a path shape; otherwise, the hinting can affect the spacing between the contours in the path's geometry and can be undesirable if you want to perform other operations such as scaling. This attribute applies to typographic shapes only; for more information see the typographic shapes chapter of <i>Inside Macintosh: QuickDraw GX Typography</i> .
<code>gxDiskShape</code>	<code>0x0200</code>	QuickDraw GX is to write this shape to disk before all shapes that do not have this attribute set when it needs to unload shapes to minimize memory requirements. The attributes <code>gxDiskShape</code> and <code>gxMemoryShape</code> are exclusive; do not set them both.
<code>gxMemoryShape</code>	<code>0x0400</code>	QuickDraw GX is to keep this shape loaded in memory as long as possible. When this attribute is set, QuickDraw GX writes this shape out to disk after all shapes are written that do not have this attribute set. The attributes <code>gxMemoryShape</code> and <code>gxDiskShape</code> are exclusive; do not set them both.

Default Shapes

When you first create a shape of a given shape type, QuickDraw GX provides an initial value for each property; those initial values define the starting characteristics of the shape. The shape QuickDraw GX creates is a copy of the default shape for that shape type (such as a line, rectangle, or glyph). There is one default shape for each shape type. These are the default properties:

- n No geometry. All applicable values and counts are set to 0.
- n A shape fill that depends on the shape type:
 - n The default empty shape has no fill.
 - n The default point, line, and curve shapes have open-frame fill.
 - n The default rectangle, polygon, path, full, bitmap, and picture shapes have even-odd fill.
 - n The default text, glyph, and layout shapes have winding fill.

Shape Objects

- n A `nil` style reference, which is equivalent to a reference to the default style object. See the chapter “Style Objects” in this book for a description of the default style object. Graphic shapes all share a single common default style; each different type of typographic shape (text, glyph, and layout) uses its own default style.
- n A `nil` ink reference, which is equivalent to a reference to the default ink object. See the chapter “Ink Objects” in this book for a description of the default ink object. All shapes except bitmaps share a common default ink object.
- n A `nil` transform reference, which is equivalent to a reference to the default transform object. See the chapter “Transform Objects” in this book for a description of the default transform object. All shapes share a common default transform, with the exception of the picture shape, which has its own default transform.
- n No attributes set (except for bitmap, picture, and layout shapes, which have the `gxMapTransformShape` attribute set).
- n An owner count of 1.
- n An empty tag list.

After creating the shape, you can change its characteristics to customize it; for example, you can give it a specific geometry. Or, if you want to create several shapes with the same customized characteristics, you can change the default shape itself to suit your purposes. If you do this, each shape that you create thereafter has the customized characteristics. See the section “Getting and Setting the Default Shape Objects” beginning on page 2-23, and the section “Resetting a Shape Object’s Properties to Their Default Values” beginning on page 2-31, for more information.

Modifying Shape Properties

After you have created a shape of a given type, you can set its various properties in order to make it useful for your purposes. Some generally applicable property-setting functions, such as those that modify the attributes or owner count, are described in this chapter. Others, more specific to individual shape types, are described in the appropriate chapters of *Inside Macintosh: QuickDraw GX Graphics* and *Inside Macintosh: QuickDraw GX Typography*.

In addition, however, QuickDraw GX provides several general-purpose functions that directly affect the geometry or type of a shape. The functions of this type described in this chapter allow you to

- n copy the geometry from one shape into another, which can have the effect of changing its type (see “Copying the Geometry From One Shape to Another” beginning on page 2-29)
- n directly manipulate shape geometry in QuickDraw GX memory (see “Directly Manipulating a Shape’s Geometry” beginning on page 2-34)
- n convert a shape of one type, such as a rectangle, to another, such as a line or a bitmap (see “Converting Shapes From One Type to Another” beginning on page 2-32)

Shape Objects

The functions that convert from one shape type to another are described in this chapter, but the rules for and consequences of conversion among shape types are specific to each shape type and thus are not described here. Table 2-5 on page 2-33 lists the chapters of *Inside Macintosh: QuickDraw GX Graphics* and *Inside Macintosh: QuickDraw GX Typography* where you can find this information.

Drawing Shapes

Two of the most fundamental and common operations you perform on shapes are drawing and its complement, hit-testing (interpreting mouse clicks or otherwise relating coordinate position to shape geometry).

You can draw a shape as soon as you have created it and set its properties (and those of its related objects). The view ports listed in the transform object associated with the shape determine where the drawn shape appears. Drawing takes into account all the information in the shape's transform, ink, style, and shape objects. This chapter describes the basic drawing function that can draw any kind of shape. See "Drawing Shapes" beginning on page 2-35 for more description and an example of drawing.

Hit-Testing Shapes

Hit-testing is the process of converting a point in the displayed representation of a shape to a location in the shape object's geometry. You can use hit-testing for shape selection, highlighting, or positioning the caret in text.

When you hit-test a shape, you can in most cases determine which part of a shape's geometry corresponds (within a certain **tolerance**, or distance) to the point you are testing against. For example, you can tell if a point is exactly on a line, or only close to it; and you can tell which edge of which glyph in a line of text is closest to the hit point.

QuickDraw GX provides a general hit-testing capability for all shapes, a specialized hit-test for testing picture shapes, another specialized hit-test for use with layout shapes, and another specialized hit-test for comparing shapes to specific pixels on a display device. See "Hit-Testing Shapes" beginning on page 2-36 for more information on the specific functions.

When you use the general hit-testing function, it returns one or more shape parts, which specify the parts of the shape's geometry corresponding to the hit point. The parts of a shape's geometry for which you can hit-test depend on the kind of shape. For example, for a typographic shape, the possible parts include those shown on the left side of Figure 2-5:

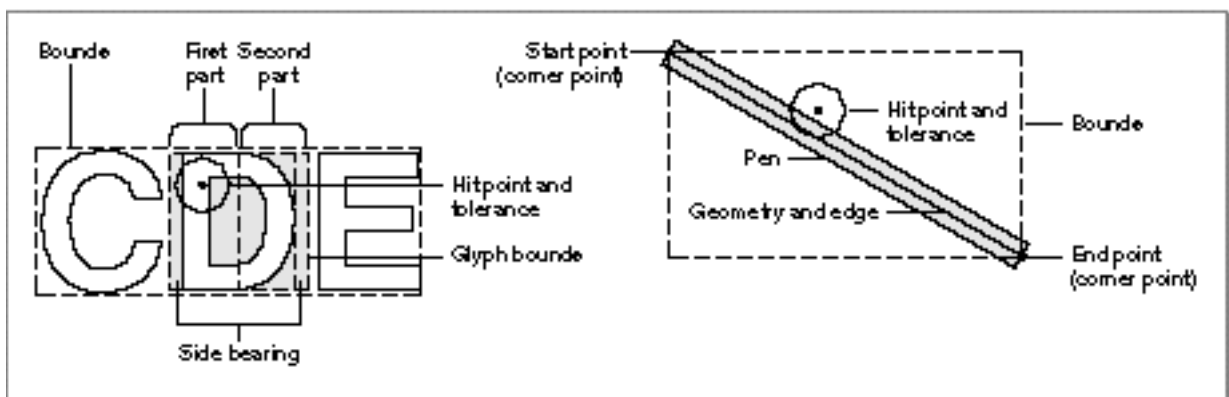
- n **bounds:** the bounding rectangle enclosing the entire typographic shape
- n **glyph bounds:** the bounding box for an individual glyph
- n **glyph first part:** the left half of the glyph
- n **glyph second part:** the right half of the glyph
- n **side bearing:** the space on either side of the glyph

Shape Objects

As another example, the possible parts for a line include those shown on the right side of Figure 2-5:

- n bounds: the bounding rectangle enclosing the start and end points of the line
- n edge: the start and end points and all the points between them on the line
- n pen: a polygon with half the width of the pen on each side of the line
- n geometry: the line's edge plus all area enclosed by it (in this case none, because a line encloses no area)

Figure 2-5 Shape parts for hit-testing



The shape parts that you can test for are defined in the `gxShapeParts` enumeration, shown on page 2-37 and described in more detail in the chapter “Transform Objects” in this book. Before performing the hit-test, you set up—in the transform object of the shape you are testing—a mask structure that defines all the shape parts that you want to test for. QuickDraw GX tests only for those parts that you specify in the shape parts mask.

For example, if the hit point on the right side of Figure 2-5 is within the tolerance of the geometry part, the function will determine that it corresponds to the bounds, the geometry, the pen, and the edge. If you want to test for geometry alone, then, you could exclude all but geometry from the test. For hit-testing the text on the left side of Figure 2-5, you might be interested only in whether the hit is within the bounding rectangle of the shape and which side of which glyph it corresponds to, so you can specify the shape parts appropriately.

When you set up the hit-test parameters, you also specify a tolerance. Tolerance is a distance (in units of geometry space), and it defines a circular area centered on the hit point. Any part that falls within that area is considered to correspond to the hit point.

Saving and Restoring Shapes

In memory, a QuickDraw GX shape consists of a shape object and (by reference) several other objects, including a style, an ink, and a transform. The locations and internal structures of those objects are private.

If you need to save a shape in a document or other external storage form, or transmit it across a network, or otherwise preserve its information in a public format, you can convert, or **flatten**, its object-based description into a stream-based description. Conversely, you can restore the object-based description of an object from its flattened form.

The flattened, stream-based format for most objects is documented in the stream format chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*. (Fonts have their own flattened format; see the font objects chapter of *Inside Macintosh: QuickDraw GX Typography* for more information.) How to flatten and unflatten shape objects is described in the section “Flattening and Unflattening Shapes” beginning on page 2-39 in this chapter.

Using Shape Objects

This section describes the basic shape-creation and shape-manipulation capabilities that QuickDraw GX provides, capabilities that are independent of the specific type of shape involved. For detailed information on using shapes of specific types, see the appropriate chapters of *Inside Macintosh: QuickDraw GX Graphics* and *Inside Macintosh: QuickDraw GX Typography*.

This section describes how you can

- n create and manipulate shape objects
- n manipulate shape object properties
- n convert shapes from one type to another
- n directly manipulate shape geometry
- n flatten and unflatten shapes
- n draw and hit-test shapes

Creating and Manipulating Shape Objects

This section describes how you can create and interact with shape objects as whole entities—to create, dispose of, copy, compare, clone, cache, load, and unload them. It also describes how to manipulate the default shapes. Manipulating the *properties* of shapes is described under “Manipulating Shape Object Properties” beginning on page 2-28.

Getting and Setting the Default Shape Objects

QuickDraw GX defines a default shape object for each shape type. These defaults are the templates QuickDraw GX uses when creating new shape objects, and you can change them to suit your purposes. Note, however, that changing the geometry for a default shape has no effect when subsequent shapes are created from the default one. A newly created shape never contains a geometry.

You can use the `GXGetDefaultShape` function to examine one of the default shape objects and the `GXSetDefaultShape` function to replace one of the default shape objects.

The properties common to all default shape objects are described under “Default Shapes” on page 2-18. Default properties specific to graphic or typographic shapes are described in *Inside Macintosh: QuickDraw GX Graphics* and *Inside Macintosh: QuickDraw GX Typography*, respectively.

The following code fragment uses `GXGetDefaultShape` to change the characteristics of the ink object referenced by the default line shape. The code obtains a reference to the default shape, and creates a temporary ink reference (`tempInk`) to the shape’s ink object. It changes the temporary ink’s color and transfer mode (with library functions `SetInkCommonColor` and `SetInkCommonTransfer`), and then assigns the modified ink back to the default shape:

```
tempInk = GXCopyToInk(nil, GXGetShapeInk
                    (GXGetDefaultShape(gxLineType)));
SetInkCommonColor(tempInk, gxBlack);
SetInkCommonTransfer(tempInk, gxXorMode);
GXSetShapeInk(GXGetDefaultShape(gxLineType), tempInk);
GXDisposeInk(tempInk);
```

The code disposes of the temporary ink after assigning it to the default shape, because that temporary reference is no longer needed.

Note

If you have created a shape object, and want to restore some of its default values, you can use the `GXResetShape` function. See the section “Resetting a Shape Object’s Properties to Their Default Values” beginning on page 2-31. u

The `GXGetDefaultShape` function is described on page 2-52. The `GXSetDefaultShape` function is described on page 2-53.

Creating and Disposing of Shape Objects

QuickDraw GX provides a number of ways for you to create a new shape object. This section describes the `GXNewShape` function, which creates a copy of the default shape for the shape type you specify. You can then customize the shape using the techniques described in the section “Manipulating Shape Object Properties” beginning on page 2-28. Other ways to create and customize specific types of shape objects are described in the chapters that describe shapes in *Inside Macintosh: QuickDraw GX Graphics* and *Inside Macintosh: QuickDraw GX Typography*. Note that you can also create a new shape by copying an existing one: see the section “Copying, Comparing, and Cloning Shape Objects” beginning on page 2-25.

Before you can create a shape or any other object, you need to be in the QuickDraw GX environment. You are not required to make any calls to accomplish this, however; QuickDraw GX sets up the environment for your application when you make your first QuickDraw GX call. If you nevertheless wish to control your application’s memory use in the QuickDraw GX environment, you can use the functions `GXNewGraphicsClient` and `GXEnterGraphics`, described in the memory management chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

The following code fragment creates a rectangle shape (`rectShape`), assigns it a fill type (closed-frame fill), and assigns its ink object a gray color (using the library function `SetShapeCommonColor`):

```
rectShape = GXNewShape (gxRectangleType);
GXSetShapeFill (rectShape, gxClosedFrameFill);
SetShapeCommonColor (rectShape, gxGray);
```

The following code fragment creates a picture shape (`docPage`) to represent the page of a document that is to be printed. It sets the `gxUniqueItemsShape` shape attribute to make sure each item in the picture has a unique reference:

```
docPage = GXNewShape(gxPictureType);
GXSetShapeAttributes(docPage, gxUniqueItemsShape);
```

(Note that this method of assigning an attribute clears all other attributes, which may be undesirable. In general, you would first call `GXGetShapeAttributes`, modify the returned attributes as needed, and then call `GXSetShapeAttributes` to reassign them.)

To delete your application’s reference to a shape object, call the `GXDisposeShape` function. You must be sure to dispose of every shape that you create. For the `docPage` shape you would make this call:

```
GXDisposeShape(docPage);
```

Shape Objects

Note that calling `GXDisposeShape` for a particular shape object may or may not actually release the memory allocated for that object, depending on its owner count. `GXDisposeShape` decreases the shape object's owner count by 1; if that brings the owner count to 0, the shape is completely deleted and its memory released (and the owner count of each object that the shape object references is then decremented). See "Manipulating a Shape Object's Owner Count" on page 2-31.

The `GXNewShape` function is described on page 2-54. The `GXDisposeShape` function is described on page 2-55.

Getting the Size of a Shape Object in Memory

Although the sizes of style, ink, and transform objects are relatively constant, shape objects vary greatly in size, mostly due to the differences in their geometries. The `GXGetShapeSize` function allows you to find out how much memory a shape occupies.

The `GXGetShapeSize` function returns only the amount of memory currently being used to represent the shape. Because QuickDraw GX can automatically unload objects from memory, the size returned by `GXGetShapeSize` does not accurately reflect the size of the object if it has been unloaded. You can call the `GXLoadShape` function before calling `GXGetShapeSize` to get a more accurate size, if necessary.

The `GXGetShapeSize` function is described on page 2-56.

Copying, Comparing, and Cloning Shape Objects

You can use the `GXCopyToShape` and `GXCopyDeepToShape` functions to copy all of the information from one shape to another or to create a new copy of a shape. The two functions are identical except that `GXCopyDeepToShape` copies more information for these shape types: for bitmap shapes, it also copies the pixel image; for picture shapes, it makes a new copy of each shape in the picture; and for glyph and layout shapes, it copies the style list.

The following code fragment copies a shape to make a version having special visual characteristics. It makes a temporary shape (`tempTextShape`) that is a copy of a text shape (`textShapeFromPicture`) within a picture shape representing a document page. The `GXCopyDeepToShape` function is not needed in this case because a text shape, unlike a glyph shape or layout shape, cannot have a style list to copy. The code doubles the size of the text and moves it by 100 points vertically before inserting it back into the page and disposing of the temporary reference.

Shape Objects

Note that this code makes use of the QuickDraw GX `ff` macro, a shorthand version of the `IntToFixed` macro. Both functions are described in the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

```
GXGetPictureParts(thePage, 2, 1, &textShapeFromPicture,
                 nil, nil, nil);
tempTextShape = GXCopyToShape (nil, textShapeFromPicture);

GXScaleShape(tempTextShape, ff(2), ff(2), 0, 0);
GXMoveShape(tempTextShape, 0, ff(100));

GXSetPictureParts(thePage, 3, 0, 1, &tempTextShape,
                 nil, nil, nil);
GXDisposeShape(tempTextShape);
```

You can test if two shape references refer to the same shape object by simply testing the references for equality. You can also compare two different shape objects for equality with the `GXEqualShape` function. For two shapes to be equal, their fill properties must be equal and their geometries must be identical. See the `GXEqualInk`, `GXEqualStyle`, and `GXEqualTransform` function descriptions in the chapters “Ink Objects,” “Style Objects,” and “Transform Objects,” respectively, in this book for the requirements for equality. Shape copies created by `GXCopyToShape` or `GXCopyDeepToShape` are always equal to the shape from which they were copied.

Equivalent geometries are not identical

Some shapes have equivalent, but not identical, geometries, and are thus not considered equal by `GXEqualShape`. For example, two polygons might have identical geometries, except that one has a duplicate point at one of its corners. The shapes are equivalent in form, but their geometries are not identical. You can remove such duplicate points with the `GXReduceShape` function, described in the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*. \cup

In certain circumstances, you may want to copy a reference to a shape object without actually copying the shape object. For example, you may want two variables to refer to the same shape object, so that editing one of them affects both. This is called **cloning** a shape, rather than copying a shape. You can use the `GXCloneShape` function to clone a shape object.

Functionally, `GXCloneShape` does nothing more than increase the owner count of a shape object. For more information about cloning objects, see the chapter “Introduction to Objects” in this book. For information on manipulating shape owner counts, see the section “Manipulating a Shape Object’s Owner Count” beginning on page 2-31 of this chapter.

The `GXCopyToShape` function is described on page 2-57. The `GXCopyDeepToShape` function is described on page 2-58. The `GXEqualShape` function is described on page 2-60. The `GXCloneShape` function is described on page 2-61.

Caching Shape Objects

Before QuickDraw GX draws any shape, it first performs some preliminary calculations on the shape's data (such as finding the shape's bounds) and stores the information in a shape cache.

In certain circumstances, you can improve the way drawing occurs on the screen by requesting that QuickDraw GX create the caches before you actually draw the shapes. For example, if you are drawing many shapes at once, you can cache all of the shapes before you draw any of them. In this way, you can minimize the amount of time between the appearance of the first shape and the completion of the last shape.

You can use the `GXCacheShape` function to create caches before drawing and you can use the `GXDisposeShapeCache` function to release the memory held by a shape cache. The `GXGetShapeCacheSize` function returns information about the size of the cache in memory.

The `GXCacheShape` function works somewhat differently from the `gxCachedShape` attribute (see Table 2-4 on page 2-16). Setting the `gxCachedShape` attribute causes QuickDraw GX to cache and predraw a shape into a compressed offscreen bitmap the first time it is drawn. Then, when you call `GXDrawShape`, the predrawn shape is simply transferred to the screen. Setting the `gxCachedShape` attribute causes very fast drawing but may greatly increase the memory required to store a shape, especially for large shapes. Calling `GXCacheShape` does not increase the memory required to draw a shape. For the fastest possible drawing (but the slowest preparation for drawing), set the `gxCachedShape` attribute and also call `GXCacheShape` before drawing.

You are not required to use any of the functions in this section. QuickDraw GX automatically creates shape caches when you draw a shape and automatically deletes shape caches when memory is low. You only need to use these functions when you want to improve your application's drawing speed.

The `GXCacheShape` function is described on page 2-62; The `GXDisposeShapeCache` function is described on page 2-63; The `GXGetShapeCacheSize` function is described on page 2-64.

Loading and Unloading Shape Objects

Although you rarely need to, you can influence memory-allocation decisions involving objects that you have created. If your application needs to have a shape object in memory, it can force QuickDraw GX to load it into memory. When your application no longer needs the shape object in a loaded state, it can instruct QuickDraw GX to unload it.

You call the `GXLoadShape` function to make sure that a shape object is in memory; if necessary, QuickDraw GX brings the object into memory from an unloaded state. You can call the `GXUnloadShape` function to instruct QuickDraw GX that it is free to unload the shape object at any time.

Shape Objects

Rather than explicitly instructing QuickDraw GX to load or unload an object, you can also set either the `gxDiskShape` or the `gxMemoryShape` attribute for the shape, which permanently affects the priority with which QuickDraw GX loads or unloads the shape. Shape attributes are described in Table 2-4 on page 2-16.

The `GXLoadShape` and `GXUnloadShape` functions are described in the memory management chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

Manipulating Shape Object Properties

This section describes how to manipulate the properties of shape objects, including those that are references to other objects. In most cases, a pair of functions respectively get and set a property. You call the `GXGetShapeProperty` function to get a copy of the shape property you need; you call the `GXSetShapeProperty` function to assign a value to a property.

For manipulating shape objects as a whole, see “Creating and Manipulating Shape Objects” beginning on page 2-22.

Getting and Setting a Shape Object’s Type, Fill, and Attributes

The functions described in this section get and set shape properties that are numerical values.

You can use the `GXGetShapeType` function to find the shape type of an existing shape, and the `GXSetShapeType` function to convert an existing shape from one shape type to another. The section “Converting Shapes From One Type to Another” beginning on page 2-32 summarizes the kinds of shape conversions QuickDraw GX supports. Beyond that section and the descriptions in Table 2-1 on page 2-9, this book does not discuss specific shape types. See *Inside Macintosh: QuickDraw GX Typography* for more information on the typographic shape types—text, glyph, and layout. (Note that `GXSetShapeType` even allows you to convert typographic shapes to graphic shapes of certain types.) See *Inside Macintosh: QuickDraw GX Graphics* for more information on graphic shape types.

The following code fragment determines the number of items (`numParts`) in a picture shape (`theShape`). The code uses `GXGetShapeType` to screen out any shape that is not a picture shape:

```
typeOfShape = GXGetShapeType(theShape);
if (typeOfShape == gxPictureType)
    numParts = GXGetPicture(theShape, nil, nil, nil, nil);
```

You can use the `GXGetShapeFill` function to find the fill of an existing shape, and the `GXSetShapeFill` function to set the fill of a shape when you create or modify it. Beyond the descriptions in Table 2-2 on page 2-13, this book does not discuss specific shape fills. See *Inside Macintosh: QuickDraw GX Typography* and *Inside Macintosh: QuickDraw GX Graphics* for more information on the valid typographic and graphic shape fills.

Shape Objects

You can use the `GXGetShapeAttributes` function to find the attributes of an existing shape and the `GXSetShapeAttributes` function to set the attributes of a shape. Shape attributes are described in the section “Shape Attributes” beginning on page 2-16.

The following code fragment is a drawing loop that rotates a text shape (`theText`) six times around the point (`x`, `y`) by 15 degrees each time, and adds the shape to a picture (`gthePage`) after each rotation. (It also changes the color at each rotation, for better visibility of the overlapping text.) The loop sets the `gxMapTransformShape` attribute of the shape, which assures that the shape geometry itself is not affected by the rotation, and thus there is no loss of precision in the geometry with repeated rotations:

```
GXSetShapeAttributes(theText, gxMapTransformShape);
for (loop = 0; loop < 6; loop++)
{
    GXSetShapeColor(theText, &textColor);
    GXRotateShape(theText, ff(15), x, y);
    GXSetPictureParts(gthePage, 0, 0, 1, &theText, nil, nil, nil);
    textColor.element.hsv.hue += 0x0940;
}
```

Note that the `gxUniqueItemsShape` attribute of `gthePage` must be set for this to work.

You can use `GXGetShapeAttributes` in combination with the `GXSetShapeAttributes` function to set and clear single attribute flags. For example, to clear the `gxDiskShape` attribute of a shape referenced by the variable `target`, you could use the following code:

```
GXSetShapeAttributes(target,
    GXGetShapeAttributes(target) & ~gxDiskShape);
```

Conversely, to set the `gxDiskShape` attribute, you could use the following code:

```
GXSetShapeAttributes(target,
    GXGetShapeAttributes(target) | gxDiskShape);
```

The `GXGetShapeType` function is described on page 2-66. The `GXSetShapeType` function is described on page 2-66. The `GXGetShapeFill` function is described on page 2-68. The `GXSetShapeFill` function is described on page 2-69. The `GXGetShapeAttributes` function is described on page 2-74. The `GXSetShapeAttributes` function is described on page 2-74.

Copying the Geometry From One Shape to Another

Like type, fill, and attributes, geometry is a property of a shape object. However, you access and manipulate a shape’s geometry somewhat differently from other properties.

Shape Objects

The `GXSetShapeGeometry` function copies the geometry (and the shape type, if the shapes are of different types) from one shape object into another. To make the function call requires two object references, and no reference to or specification of either object's geometry. There is no associated `GXGetShapeGeometry` call. Using `GXSetShapeGeometry` is a simple way to reuse an existing shape by turning it into a copy of another shape. As with `GXSetShapeType`, this book does not discuss the specific rules for and consequences of converting one shape type to another with `GXSetShapeGeometry`. See *Inside Macintosh: QuickDraw GX Graphics* and *Inside Macintosh: QuickDraw GX Typography* for conversion information for graphic and typographic shape types.

To do more than simply copy geometries—to gain access to and actually manipulate the contents of a shape's geometry—requires another set of functions, including the `GXGetShapeStructure` function. See the section “Directly Manipulating a Shape's Geometry” beginning on page 2-34. In most situations, however, you use functions specific to a given shape type to manipulate that type of shape's geometry. Those kinds of functions are described, along with each shape type, in *Inside Macintosh: QuickDraw GX Graphics* and *Inside Macintosh: QuickDraw GX Typography*.

To copy an entire object, rather than just its geometry, you can use the `GXCopyToShape` or `GXCopyDeepToShape` functions; see “Copying, Comparing, and Cloning Shape Objects” on page 2-25.

The `GXSetShapeGeometry` function is described on page 2-67.

Getting and Setting a Shape Object's Style, Ink, and Transform

Every QuickDraw GX shape object has an associated style object, ink object, and transform object. You can use the `GXGetShapeStyle`, `GXGetShapeInk`, and `GXGetShapeTransform` functions to determine which of each type of object is referenced by a particular shape. Conversely, you can use the `GXSetShapeStyle`, `GXSetShapeInk`, and `GXSetShapeTransform` functions to change these references.

Because style objects can be shared among different QuickDraw GX shapes, the `GXGetShapeStyle` function can return a reference to the same style object for two different shapes. Likewise, the `GXGetShapeInk` and `GXGetShapeTransform` functions can return identical ink objects or transform objects for different shapes.

Calling `GXSetShapeStyle`, `GXSetShapeInk`, or `GXSetShapeTransform` increments the owner count of the specified style, ink, or transform object by 1, and disposes of the previously assigned style, ink, or transform. In certain cases, depending on how you create such an object or assign it to a shape, you may need to modify that object's owner count explicitly; see “Manipulating a Shape Object's Owner Count” on page 2-31.

The following code fragment draws a dashed version of a shape. The code first calls `GXGetShapeStyle` to obtain the style object attached to the shape `theShape`; it then clones the style and assigns a temporary reference (`saveStyle`) to the style. The code then assigns different style properties to the shape and draws it. After drawing the shape, the code restores the original style to the shape, using `GXSetShapeStyle`:

Shape Objects

```

saveStyle = GXCloneStyle(GXGetShapeStyle(theShape));
GXSetShapePen(theShape, ff(1));
GXSetShapeDash(theShape, &dash);
GXDrawShape(theShape);

GXSetShapeStyle(theShape, saveStyle);
GXDisposeStyle(saveStyle);

```

As usual, after it is finished with the temporary reference `saveStyle`, the code disposes of it. For more information and examples of cloning, see for example the discussions of owner count in the chapter “Style Objects” in this book.

The `GXGetShapeStyle` function is described on page 2-69; the `GXSetShapeStyle` function is described on page 2-70. The `GXGetShapeInk` function is described on page 2-71; the `GXSetShapeInk` function is described on page 2-71. The `GXGetShapeTransform` function is described on page 2-72; the `GXSetShapeTransform` function is described on page 2-73.

Resetting a Shape Object’s Properties to Their Default Values

When you create a new shape with the `GXNewShape` function, QuickDraw GX creates the new shape object by copying the appropriate default shape object. QuickDraw GX does not create a new style, ink, or transform object for the new shape, however. Instead, the new shape contains references to the same style, ink, and transform as the corresponding default shape. You are free to install a new style, ink, or transform in the shape using functions such as `GXSetShapeStyle`, `GXSetShapeInk`, and `GXSetShapeTransform`.

If you do install a new style, ink, or transform in a shape and you want to revert back to the default style, ink, and transform, you can use the `GXResetShape` function. This function also resets the shape’s attributes and fill properties to match the default shape, but does not alter the shape’s geometry, owner count, or tag list.

The `GXResetShape` function is described on page 2-75.

Manipulating a Shape Object’s Owner Count

The owner count of an object indicates the number of current references to that object. In general, QuickDraw GX manages owner counts for you. For example, when you create a new shape object you give it a variable name such as `myShape`. QuickDraw GX sets the owner count of the new shape to 1, because your application variable is the only current reference to the shape. As another example, when you add a shape to a picture, QuickDraw GX increments the shape’s owner count, corresponding to the new reference to the shape contained in the picture.

Shape Objects

The following code fragment is part of a routine that constructs a house image (`gOurHouse`) as a picture shape, building it out of individual geometric shapes. As each component shape (`houseBorderShape` and `doorShape`, in this fragment) is added to the picture shape, its owner count is increased; to balance that increase, and because that component shape's reference is no longer needed, it is disposed of.

```
GXSetShapeFill(houseBorderShape, gxHollowFill);
GXSetPictureParts(gOurHouse, 1, 0, 1, houseBorderShape,
                  nil, nil, nil);
GXDisposeShape(houseBorderShape);

GXSetShapeFill(doorShape, gxHollowFill);
GXSetPictureParts(gOurHouse, 1, 0, 1, doorShape,
                  nil, nil, nil);
GXDisposeShape(doorShape);
```

If you want to manage a shape's owner count directly—for example, if you want to track object references that you place in your own data structures, or if you want to know whether a shape object is shared—you can use the `GXGetShapeOwners` function to determine the owner count of a shape, and the `GXCloneShape` and `GXDisposeShape` functions to change the owner count of a shape. The `GXCloneShape` function increments the shape's owner count, and the `GXDisposeShape` function decrements the shape's owner count, freeing the memory used by the shape if the owner count goes to 0.

The `GXGetShapeOwners` function is described on page 2-76. The `GXCloneShape` function is described on page 2-61. The `GXDisposeShape` function is described on page 2-55.

Getting and Setting a Shape Object's Tag References

You can examine the list of references to tag objects currently associated with a shape using the `GXGetShapeTags` function. Once you create a tag object, you can attach it to a shape object using the `GXSetShapeTags` function. You can attach as many tag objects as you like to a shape object.

Tag objects and the basic functions for manipulating them are described in the chapter “Tag Objects” in this book. That chapter also lists the common tag types defined and reserved by Apple Computer, Inc.

The `GXGetShapeTags` function is described on page 2-77. The `GXSetShapeTags` function is described on page 2-78.

Converting Shapes From One Type to Another

QuickDraw GX allows you to change the types of the shape objects you have created. You use the `GXGetShapeType` function, described on page 2-66 of this chapter, to determine the type of a shape. To convert a shape to a new type, you use the `GXSetShapeType` function, described on page 2-66 of this chapter.

Shape Objects

The rules for conversion among shape geometries are specific to each shape type and thus are not described here. See the appropriate chapters of *Inside Macintosh: QuickDraw GX Graphics* and *Inside Macintosh: QuickDraw GX Typography* for this information. Table 2-5 describes where to look in each book for information regarding each possible kind of conversion.

Table 2-5 Where to find information on shape-type conversion

	To a geometric shape	To a bitmap shape	To a picture shape	To a typographic shape
From a geometric shape	See “Geometric Shapes” in <i>QuickDraw GX Graphics</i>	See “Bitmap Shapes” in <i>QuickDraw GX Graphics</i>	See “Picture Shapes” in <i>QuickDraw GX Graphics</i>	(not possible)
From a bitmap shape	See “Geometric Shapes” in <i>QuickDraw GX Graphics</i>	(no change)	See “Picture Shapes” in <i>QuickDraw GX Graphics</i>	(not possible)
From a picture shape	See “Geometric Shapes” in <i>QuickDraw GX Graphics</i>	See “Bitmap Shapes” in <i>QuickDraw GX Graphics</i>	(no change)	(not possible)
From a typographic shape	See “Typographic Shapes” in <i>QuickDraw GX Typography</i>	See “Bitmap Shapes” in <i>QuickDraw GX Graphics</i>	See “Picture Shapes” in <i>QuickDraw GX Graphics</i>	See “Typographic Shapes” in <i>QuickDraw GX Typography</i>

Another common kind of shape conversion is not from one shape type to another, but from standard object form into primitive form. Some functions, such as `GXSetShapeClip`, described in the chapter “Transform Objects” in this book, require a primitive shape to hold the clip shape. A **primitive shape** is a shape whose stylistic information has been incorporated into the shape’s geometry. For example, a horizontal line with a thick pen style becomes a rectangle when converted to a primitive shape. To make a shape into a clip, you first convert it to its primitive form with the function `GXPrimitiveShape`. For more information about primitive shapes in general, see the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*. For information on primitive shapes for typographic shapes, and the difference between using `GXPrimitiveShape` and `GXSetShapeType` to obtain a primitive shape, see the typographic shapes chapter of *Inside Macintosh: QuickDraw GX Typography*.

Directly Manipulating a Shape's Geometry

The geometry of a shape object is its most central property. Unlike other properties, it can be made accessible to you as a structure that you can modify directly, in place in QuickDraw GX memory. QuickDraw GX provides a group of functions with which you can access a shape's geometry and notify QuickDraw GX once you have modified it. Note that in most cases you don't need to do this; QuickDraw GX provides many functions, specific to each type of shape, with which you can access and modify geometry. The functions described here are provided as an added convenience.

These functions do not provide you with information about the formats of the data structures that make up shape geometries; they simply give you a pointer to the geometry. How you manipulate that information depends on the type of shape whose geometry you are accessing. The structures of individual shape geometries are described in the shape-specific chapters of *Inside Macintosh: QuickDraw GX Graphics* and *Inside Macintosh: QuickDraw GX Typography*.

Before accessing a shape's geometry, you must set its `gxDirectShape` attribute to make sure that it is loaded into directly accessible memory. To access the geometry, you first call the `GXLockShape` function to make sure the shape object doesn't move until you are finished with it. You then call the `GXGetShapeStructure` function, which returns a pointer to the shape's geometry. You can then modify the geometry as needed. Once finished, you call `GXUnlockShape` to free the shape object for relocation in memory as needed. Finally, you must call `GXChangedShape` to notify QuickDraw GX that you have changed the geometry.

Listing 2-1 is a partial listing of a function that accesses the geometry of the path shape `myShape`, manipulating its geometry as a `gxPaths` structure in a buffer of size `size`.

Listing 2-1 Directly accessing a shape's geometry

```
.
. /* set up the shape (not shown) */
.
/* set the direct shape attribute if not set */
GXSetShapeAttributes (myShape,
                      GXGetShapeAttributes(myShape) | gxDirectShape);

/* lock and examine or change the shape */
GXLockShape(myShape);
shapeStruct = (gxPaths*)GXGetShapeStructure(myShape, &size);
.
. /* unlock the shape as soon as access no longer needed */
.
GXUnlockShape(myShape);

/* notify QuickDraw GX of a change only if geometry changed */
GXChangedShape(myShape);
```

IMPORTANT

Memory-handling complications can occur with locked objects. Locking an object fragments the QuickDraw GX heap, which can result in lower performance. Furthermore, if a fragmented-memory condition occurs during a call, QuickDraw GX may unlock all objects and restart the call. Therefore, be careful about performing memory-intensive operations while there are locked objects in QuickDraw GX memory; they may become unlocked and be moved. s

The `GXLockShape` function is described on page 2-80. The `GXGetShapeStructure` function is described on page 2-82. The `GXUnlockShape` function is described on page 2-81. The `GXChangedShape` function is described on page 2-83.

Drawing and Hit-Testing Shapes

Drawing and hit-testing are common actions you may perform with any kind of shape. The most basic QuickDraw GX drawing function is `GXDrawShape`, although there are other functions for drawing specific types of shapes. Only `GXDrawShape` is described here.

The functions you use for hit-testing are `GXHitTestShape`, `GXHitTestPicture`, `GXHitTestLayout`, and `GXHitTestDevice`. Only `GXHitTestShape` is described here.

Drawing Shapes

Drawing a shape is the logical conclusion to creating it and setting its properties. Drawing occurs in the view port or view ports specified in the transform object associated with the shape. Drawing takes into account all the information in the shape's transform, ink, style, and shape objects.

What it means to draw a specific type of shape and how changing the information in a shape alters its drawn appearance is described, along with each type of shape, in *Inside Macintosh: QuickDraw GX Graphics* and *Inside Macintosh: QuickDraw GX Typography*. Furthermore, for many shape types, QuickDraw GX provides specialized drawing functions, such as `GXDrawLine` and `GXDrawGlyphs`—described in those books—that allow you to create, draw, and dispose of an object with a single call.

At its most basic, though, creating and drawing a shape is as simple as the following listing for creating and drawing a path shape shows:

```
gxShape myShape;                               /* allocate the variable */
myShape = GXNewShape(gxPathType);              /* create the shape */
.
.                                               /* set its properties */
.
GXDrawShape(myShape);                          /* draw it */
```

The `GXDrawShape` function is described on page 2-84.

Hit-Testing Shapes

Hit-testing converts a coordinate location to a shape-geometry location. It can give you feedback on user actions involving a shape you have drawn. For example, you use hit-testing to select a shape the user has clicked the mouse over, to select a point within a shape, or to position the insertion point and draw the caret within the text of a typographic shape.

QuickDraw GX provides a general hit-testing function for all shapes, plus specialized functions for hit-testing picture shapes, layout shapes, and pixels on a display device:

- n `GXHitTestShape` tests a point in local space against a shape's geometry. The test tells you which part of a shape's geometry—out of a specified set of parts—corresponds (within the tolerance) to the point you are testing with. The `GXHitTestShape` function is described in this chapter.
- n `GXHitTestPicture` tests a point in local space against a picture shape. The test tells you which part of which shape within the picture corresponds (within the tolerance) to the point you are testing against (subject to the constraints on shape overlap and hierarchy that you provide). The `GXHitTestPicture` function is described in the picture shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*.
- n `GXHitTestLayout` tests a point in local space against the text of a layout shape. The test tells you, along with other information, which character in the text corresponds to the point. Note that you use `GXHitTestShape` to test typographic shapes other than layout shapes, and you can use it for layout shapes also; it gives different kinds of information from `GXHitTestLayout`. The `GXHitTestLayout` function is described in the layout carets, highlighting, and hit-testing chapter of *Inside Macintosh: QuickDraw GX Typography*.
- n `GXHitTestDevice` tests a pixel (a point in device space) against a shape's geometry. The test tells you whether or not any part of the shape's geometry is within a certain distance of the pixel. The `GXHitTestDevice` function is described in the chapter "View-Related Objects" in this book.

When you hit-test a shape with `GXHitTestShape`, you must first set up the shape parts mask and the tolerance, two components of the **hit-test parameters** property of a shape's transform object. You pass that information to `GXHitTestShape`, and QuickDraw GX returns information in the **hit-test info structure**.

The tolerance is a distance (in units of geometry space), and it defines a circular area centered on the hit point. Any part that falls within that area is considered to correspond to the hit point.

Shape Parts

When you use `GXHitTestShape`, it returns one or more shape parts, which specify the parts of the shape's geometry corresponding to the hit point. The parts of a shape's geometry for which you can hit-test depend on the kind of shape. The shape parts that you can test for are defined in the `gxShapeParts` enumeration. Before calling `GXHitTestShape`, you set up, in the transform object, a mask of all the shape parts that you want to test for. `GXHitTestShape` can test only for parts that you specify in the shape parts mask. These are the possible values to put into the mask:

Shape Objects

```

enum gxShapeParts {          /* (in order of evaluation) */
    gxNoPart                 = 0,
    gxBoundsPart             = 0x0001,
    gxGeometryPart           = 0x0002,
    gxPenPart                 = 0x0004,
    gxCornerPointPart        = 0x0008,
    gxControlPointPart       = 0x0010,
    gxEdgePart               = 0x0020,
    gxJoinPart               = 0x0040,
    gxStartCapPart           = 0x0080,
    gxEndCapPart             = 0x0100,
    gxDashPart               = 0x0200,
    gxPatternPart            = 0x0400,
    gxGlyphBoundsPart        = gxJoinPart,
    gxGlyphFirstPart         = gxStartCapPart,
    gxGlyphLastPart          = gxEndCapPart,
    gxSideBearingPart        = gxDashPart,
    gxAnyPart                 = gxBoundsPart | gxGeometryPart |
                                gxPenPart | gxCornerPointPart | gxControlPointPart |
                                gxEdgePart | gxJoinPart | gxStartCapPart |
                                gxEndCapPart | gxDashPart | gxPatternPart
};

typedef long gxShapePart;

```

These values are described in more detail in the chapter “Transform Objects” in this book. Note that values specifying join, cap, and dash parts in geometric shapes are used in typographic shapes to specify various glyph parts. Note also that you can specify no parts or all parts in the mask. You decide which shape parts are appropriate for your needs.

Hit-Test Info Structure

When you call `GXHitTestShape`, it returns some information as a function result and other information in a hit-test info structure. The first three fields of the hit-test info structure give all the relevant information about the hit:

```

struct gxHitTestInfo {
    gxShapePart    what;
    long           index;
    Fixed          distance;
    gxShape        which;
    gxShape        containerPicture;
    long           containerIndex;
    long           totalIndex;
};

```

Shape Objects

The `what` field tells you which shape parts out of those specified in your mask were hit, if any. It is identical to the `GXHitTestShape` function result.

The `index` field tells you the index number of the point in the geometry that is closest to the hit point.

The `distance` field tells you how far, in geometry coordinates, the hit point is from the first shape part that was hit. `GXHitTestShape` analyzes shape parts in a specific order—the order listed in the `gxShapeParts` enumeration. By carefully specifying shape parts, you can use `GXHitTestShape` to obtain specific distance information for a given part. For example, if you are hit-testing a line like that shown in Figure 2-5 on page 2-21, you can determine the distance from the hit point to the pen if you exclude both bounds and geometry from the test.

The remaining fields in the hit-test info structure are not used by `GXHitTestShape`.

Hit-Testing Example

Listing 2-2 uses hit-testing to determine whether a point (`aPoint`) is contained in the geometry that represents a shape (`gShape`). The code sets up a shape-part mask (`mask`) specifying that only the geometry it to be tested for, and calls the `GXSetShapeHitTest` function to assign the mask, plus a tolerance of zero, to the shape's transform.

Listing 2-2 Hit-testing a line

```
gxShape      pointShape;
gxPoint      aPoint = {ff(50), ff(51)};
gxShapePart  mask = gxGeometryPart;
gxShapePart  resultMask;
gxHitTestInfo resultInfo;

pointShape = GXNewPoint(&aPoint);
GXSetShapeHitTest(gShape, mask, ff(0));
resultMask = GXHitTestShape(gShape, &aPoint, &resultInfo);
GXDisposeShape(pointShape);
```

The function result from `GXHitTestShape` tells which part of the shape was hit. Because only one part (`gxGeometryPart`) is specified and tolerance is 0, a successful hit is possible only if `aPoint` is actually within the geometry of the shape.

In the event of a successful hit, `GXHitTestShape` also fills in a `gxHitTestInfo` structure (`resultInfo` parameter) that contains additional information about the hit.

The `gxHitTestInfo` structure is described on page 2-50. The `GXHitTestShape` function is described on page 2-86. Because the shape parts to test against are specified in a shape's transform object, the list of defined QuickDraw GX shape parts, and the `GXSetShapeHitTest` function, are described in the chapter "Transform Objects" in this book.

Flattening and Unflattening Shapes

In order to save a QuickDraw GX shape (shape object plus its referenced objects) to external storage, transmit it across a network, or save it to the Clipboard, you must convert it into an equivalent flattened, rather than object-based, description. The flattened information is a compressed and stream-based description with a public format so that applications can share the data and reconstruct the objects.

You can use the `GXFlattenShape` function to convert any shape (even a picture shape, which contains other shapes) into its flattened form. You can then store the data, examine it, or manipulate it as you wish; the data follows the format defined in the stream format chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

To reconstruct a shape's object-based description from its flattened stream, you can manually create and initialize a set of objects based on the information in the stream, but if QuickDraw GX is available, it is far easier and more efficient to use the `GXUnflattenShape` function to do it automatically.

To use the flattening or unflattening functions, you first allocate a structure called a **spool block**. The spool block contains needed information and points to a buffer that holds the flattened data. In the spool block, you are required to provide a pointer to a callback spool function that you provide. The spool function reads the stream data into the buffer or writes it to a file from the buffer.

Listing 2-3 is a library function that flattens a shape and returns a handle to the flattened data. It uses a spool-block structure (`spool`) embedded within a library-defined structure (`block`) of type `UserSpool`. The function sets up the spool-block structure, including placing into it a pointer to the spool function. It specifies `nil` for the buffer pointer and 0 for the buffer size, in which case QuickDraw GX allocates a default buffer for the task. When it calls `GXFlattenShape`, the function sets two flatten flags, so that both a list of fonts and a list of all the individual glyphs used is attached to the flattened shape.

Listing 2-3 Flattening a shape

```
Handle ShapeToHandle(gxShape source)
{
    UserSpool block;

    block.spool.spoolProcedure = (long (*)(gxSpoolCommand,
                                         struct gxSpoolBlock *)) HandleSpoolProc;
    block.spool.buffer = nil;
    block.spool.bufferSize = 0;
    GXFlattenShape(source,
                   gxFontListFlatten | gxFontGlyphsFlatten,
                   &block.spool);
    return block.data;
}
```

Listing 2-4 is a library function that unflattens a shape from data referenced by a handle (`source`). Like Listing 2-3, it sets up a spool-block structure and places into it a pointer to the spool function. When it calls `GXUnflattenShape`, the function specifies the size of the flattened data and the list of view ports to be assigned to the unflattened shape's transform object.

Listing 2-4 Unflattening a shape

```
gxShape HandleToShape(Handle source, long count,
                    const gxViewPort portList[])
{
    UserSpool block;

    block.spool.spoolProcedure = (long (*)(gxSpoolCommand,
                    struct gxSpoolBlock *)) HandleSpoolProc;
    block.spool.buffer = nil;
    block.spool.bufferSize = 0;
    block.data = source;
    return GXUnflattenShape(&block.spool, count, portList);
}
```

Your flattening/unflattening spool function responds to five commands from QuickDraw GX (described on page 2-92). In most cases it simply reads or writes a buffer of data at a time during the flattening or unflattening operation, and then closes up when the operation is finished. However, for special purposes you can write a spool function that parses the stream of data by reading information in the spool block and manipulating the size of the buffer that QuickDraw GX can read from or write into.

Listing 2-5 is a partial listing that shows the overall structure of a typical spool function for flattening and unflattening. This function, however, parses the stream as it is being flattened or unflattened. In the case of writing (flattening), the listing shows that the function sets the buffer size to equal the current operation size so that no more than a single operation can be flattened at once. Therefore, each time it is called, the spool file can read the fields of the spool block to determine the kind of information the current operation consists of and decide how large to make the buffer for the next write.

Listing 2-5 A spool function that parses shape data

```

static long MyParseSpoolProc(spoolCommand command,
                             gxSpoolBlock *block)
{
    switch (command) {
        case openReadSpool:
            .
            . /* spool function prepares for unflattening */
            .
            break;
        case openWriteSpool:
            .
            . /* spool function prepares for flattening */
            .
            break;
        case closeSpool:
            .
            . /* spool function closes up when finished */
            .
            break;
        case readSpool:
            .
            . /* spool function parses and reads for unflattening */
            .
            break;
        case writeSpool:

            /* see if current operation < 32K (real buffer size) */
            if (block->spool.operationSize < 32768)

                /* set buffer size to operation size */
                block->spool.bufferSize = block->spool.operationSize;
            else
                block->spool.bufferSize = 32000; /* don't overflow */
            . /*
            .   Spool function examines spool block, parses data,
            .   writes flattened data to disk
            .   */
            break;
    }
}

```

Shape Objects

The application sets up the conditions for this spool function by first allocating a 32 KB buffer, but setting the `size` field of the spool block to 1. This causes `GXFlattenShape` or `GXUnflattenShape` to read only a single byte into the buffer the first time through, after which the spool function can analyze that byte and proceed with parsing. (For simple reading or writing, your application typically sets the `size` field to the actual size of the buffer—32 KB in this case—and the spool function does not parse the stream at all).

The `GXFlattenShape` function is described on page 2-88. The `GXUnflattenShape` function is described on page 2-90. The spool block structure is described on page 2-49. The application-defined spool function is described on page 2-91. The flatten flags are described on page 2-48.

Shape-Related Functions Described Elsewhere

Table 2-6 lists every QuickDraw GX function whose name contains the word *Shape*, but whose description is not found in this chapter. For each book and chapter, the table lists the shape-related functions described in that chapter. Table 2-6 is intended to help you locate the descriptions of functions you may have been searching for in this chapter.

Table 2-6 Shape-related functions described elsewhere

Book and chapter	Shape functions described
<i>Inside Macintosh: QuickDraw GX Objects</i> [this book]	
“Ink Objects”	GXGetShapeColor GXSetShapeColor GXGetShapeInkAttributes GXSetShapeInkAttributes GXGetShapeTransfer GXSetShapeTransfer
“Transform Objects”	GXGetShapeClip GXSetShapeClip GXGetShapeHitTest GXSetShapeHitTest GXGetShapeMapping GXSetShapeMapping GXGetShapeViewPorts GXSetShapeViewPorts GXMapShape GXMoveShape GXMoveShapeTo GXRotateShape GXScaleShape GXSkewShape

Table 2-6 Shape-related functions described elsewhere (continued)

Book and chapter	Shape functions described
“View-Related Objects”	GXGetShapeDeviceArea GXGetShapeDeviceBounds GXGetShapeDeviceColors GXGetShapeGlobalBounds GXGetShapeGlobalViewPorts GXGetShapeGlobalViewDevices GXGetShapeLocalBounds
<i>Inside Macintosh: QuickDraw GX Graphics</i>	
“Geometric Shapes”	GXCountShapeContours GXCountShapePoints GXGetShapeFill GXSetShapeFill GXGetShapeIndex GXGetShapeParts GXSetShapeParts GXGetShapePoints GXSetShapePoints GXNewShapeVector GXSetShapeVector
“Geometric Styles”	GXGetShapeCap GXSetShapeCap GXGetShapeCurveError GXSetShapeCurveError GXGetShapeDash GXSetShapeDash GXGetShapeDashPositions GXGetShapeJoin GXSetShapeJoin GXGetShapePattern GXSetShapePattern GXGetShapePatternPositions GXGetShapePen GXSetShapePen GXGetShapeStyleAttributes GXSetShapeStyleAttributes

continued

Table 2-6 Shape-related functions described elsewhere (continued)

Book and chapter	Shape functions described
“Geometric Operations”	GXContainsBoundsShape GXContainsShape GXDifferenceShape GXExcludeShape GXGetShapeArea GXGetShapeBounds GXSetShapeBounds GXGetShapeCenter GXGetShapeDirection GXGetShapeLength GXInsetShape GXIntersectShape GXInvertShape GXReduceShape GXReverseDifferenceShape GXReverseShape GXShapeLengthToPoint GXSimplifyShape GXTouchesBoundsShape GXTouchesShape GXUnionShape
“Bitmap Shapes”	GXGetShapePixel GXSetShapePixel
<i>Inside Macintosh: QuickDraw GX Typography</i>	
“Typographic Styles”	GXGetShapeDeviceFontMetrics GXGetShapeEncoding GXSetShapeEncoding GXGetShapeFace GXSetShapeFace GXGetShapeFont GXSetShapeFont GXGetShapeTextSize GXSetShapeTextSize GXGetShapeJustification GXSetShapeJustification GXGetShapeFontMetrics GXGetShapeFontVariations GXSetShapeFontVariations GXGetShapeFontVariationSuite GXGetShapeLocalFontMetrics GXGetShapeTextAttributes GXSetShapeTextAttributes GXGetShapeTypographicBounds

Table 2-6 Shape-related functions described elsewhere (continued)

Book and chapter	Shape functions described
“Layout Shapes”	GXGetLayoutShapeParts GXSetLayoutShapeParts
“Layout Styles”	GXGetShapeRunControls GXSetShapeRunControls GXGetShapeRunFeatures GXSetShapeRunFeatures GXGetShapeRunGlyphSubstitutions GXSetShapeRunGlyphSubstitutions GXGetShapeRunKerningAdjustments GXSetShapeRunKerningAdjustments
“Layout Line Control”	GXGetShapeRunGlyphJustOverrides GXSetShapeRunGlyphJustOverrides GXGetShapeRunPriorityJustOverride GXSetShapeRunPriorityJustOverride
<i>Inside Macintosh: QuickDraw GX Environment and Utilities</i>	
“QuickDraw GX Debugging”	GXGetShapeDrawError GXValidateShape

Shape Objects Reference

This section provides reference information about the data structures and functions that allow you to create and manipulate shape objects and alter their properties. It includes

- n type definitions of the data types, including enumerations, that are specific to shape objects
- n descriptions of the QuickDraw GX functions that operate on shape objects in general, independent of the type of shape involved
- n a description of an application-defined function used for flattening and unflattening shapes

Constants and Data Types

This section describes the constants and the data types that you use to obtain and provide information about shape objects.

The Shape Object

QuickDraw GX provides you with access to an individual shape object through a `gxShape` reference:

```
typedef struct gxPrivateShapeRecord *gxShape;
```

In this type definition, `gxShape` is a type-checked reference, not an actual pointer to any defined structure. The contents of the shape object are private.

Shape Type

A shape object's shape type specifies what type of geometry the shape object has. Constants for all shape types are defined in the `gxShapeTypes` enumeration:

```
enum gxShapeTypes {
    gxEmptyType = 1,
    gxPointType,
    gxLineType,
    gxCurveType,
    gxRectangleType,
    gxPolygonType,
    gxPathType,
    gxBitmapType,
    gxTextType,
    gxGlyphType,
    gxLayoutType,
    gxFullType,
    gxPictureType
};

typedef long gxShapeType;
```

The individual shape types are described further in Table 2-1 on page 2-9.

Shape Fill

Each shape object has a shape fill property. The shape fill specifies how QuickDraw GX interprets the geometry of the shape: how the shape is drawn, how the shape is hit-tested, and how certain geometric operations, like the intersection operation, interpret the shape.

Shape Objects

Constants for all shape fills are defined in the `gxShapeFills` enumeration:

```
enum gxShapeFills {
    gxNoFill,                /* shape not drawn */
    gxOpenFrameFill,        /* framed, one edge left open */
    gxFrameFill             = gxOpenFrameFill,
    gxClosedFrameFill,     /* framed, closed completely */
    gxHollowFill           = gxClosedFrameFill,
    gxEvenOddFill,        /* filled using even-odd rule */
    gxSolidFill            = gxEvenOddFill,
    gxWindingFill,        /* filled using winding-number rule */
    gxInverseEvenOddFill, /* filled inverse of even-odd rule */
    gxInverseSolidFill     = gxInverseEvenOddFill,
    gxInverseFill          = gxInverseEvenOddFill,
    gxInverseWindingFill   /* filled inverse of winding-number */
};

typedef long gxShapeFill;
```

The individual shape fills are described further in Table 2-2 on page 2-13.

Shape Attributes

Each shape object has a set of attributes. Shape **attributes** are a group of flags that modify the behavior of the shape object. Constants for all shape attributes are defined in the `gxShapeAttributes` enumeration:

```
enum gxShapeAttributes {
    gxNoAttributes,                /* no attributes set */
    gxDirectShape                  = 0x0001, /* prefer GX heap */
    gxRemoteShape                  = 0x0002, /* prefer accel. memory */
    gxCachedShape                  = 0x0004, /* optimize drawing */
    gxLockedShape                  = 0x0008, /* lock shape geometry */
    gxGroupShape                   = 0x0010, /* treat as single shape */
    gxMapTransformShape            = 0x0020, /* alter transform */
    gxUniqueItemsShape             = 0x0040, /* copy picture items */
    gxIgnorePlatformShape          = 0x0080, /* use glyph codes */
    gxNoMetricsGridShape           = 0x0100, /* don't use hinting */
    gxDiskShape                    = 0x0200, /* unload this first */
    gxMemoryShape                  = 0x0400  /* unload this last */
};

typedef long gxShapeAttribute;
```

The individual shape attributes are described further in Table 2-4 on page 2-16.

Flatten Flags

The flatten flags are used in a parameter to the `GXFlattenShape` function, to control the amount of font and bitmap information to include in a flattened shape. The flatten flags are defined in the `gxFlattenFlags` enumeration:

```
enum gxFlattenFlags {
    gxFontListFlatten      = 0x01,
    gxFontGlyphsFlatten   = 0x02,
    gxFontVariationsFlatten = 0x04,
    gxBitmapAliasFlatten   = 0x08
};
```

```
typedef long gxFlattenFlag;
```

Constant descriptions

`gxFontListFlatten`

Instructs the `GXFlattenShape` function to attach to the flattened shape a tag object containing a list of the fonts referenced in the shape.

`gxFontGlyphsFlatten`

Instructs the `GXFlattenShape` function to attach to the flattened shape a tag object containing a list of the specific glyphs used from each font referenced by the shape.

`gxFontVariationsFlatten`

Instructs the `GXFlattenShape` function to attach to the flattened shape a tag object containing variation-axis coordinates describing all font variations used by the flattened shape.

`gxBitmapAliasFlatten`

Instructs the `GXFlattenShape` function to include with the flattened shape all image data from any bitmap shapes that are referenced by the shape. If this flag is not set, image data from bitmap shapes whose image data is disk-based is not included in the flattened shape, although the image data is not lost because a tag object specifying the file holding the image data is flattened along with the shape.

For more information on flattening shapes, see “Flattening and Unflattening Shapes” beginning on page 2-39. The `GXFlattenShape` function is described on page 2-88. For information on font variations, see the font objects chapter of *Inside Macintosh: QuickDraw GX Typography*. For information on bitmap image data, see the bitmap shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*.

The Spool Block

The spool block structure is set up by an application before calling `GXFlattenShape` or `GXUnflattenShape`. Both the application and QuickDraw GX use and place values into the spool block.

```
struct gxSpoolBlock {
    gxSpoolProcPtr    spoolProcedure;
    void              *buffer;
    long              bufferSize;
    long              count;
    long              operationSize;
    long              operationOffset;
    gxGraphicsOpcode lastTypeOpcode;
    gxGraphicsOpcode currentOperation;
    gxGraphicsOpcode currentOperand;
    unsigned char     compressed;
};
```

Field descriptions

`spoolProcedure` A pointer to an application-defined function that either saves the flattened data or supplies the data for unflattening. The `gxSpoolProcPtr` type is defined as follows:

```
typedef long (*gxSpoolProcPtr)
             (gxSpoolCommand command,
              struct gxSpoolBlock *block);
```

The format for the spool function is described on page 2-91.

`buffer` A pointer to a buffer that holds the flattened data, after flattening or before unflattening. In either case the buffer is allocated by the application.

`bufferSize` The size of the buffer. (Set by the application.)

`count` The number of bytes of data read into or out of the buffer. (Set by QuickDraw GX.)

`operationSize` The size of the current operation in the flattened stream. It is equal to the size field of the operand of the current operation. For flattening, it is the amount of data that QuickDraw GX will place into the buffer to complete the current operation; for unflattening, it is the amount of information that the spool function must place in the buffer to complete the current operation. (Set by QuickDraw GX.)

Shape Objects

operationOffset

For flattening, the offset in bytes from the beginning of the current operation to the end of the data currently in the buffer. For unflattening, the offset in bytes from the beginning of the current operation to the start of the data that needs to be placed in the buffer. It is the amount of the current operation that has so far been flattened or is about to be unflattened. (Set by QuickDraw GX.)

lastTypeOpcode

The type of object currently being flattened or unflattened. It is one of the constants defined in the `gxGraphicsNewOpcode` enumeration. (Set by QuickDraw GX.)

currentOperation

The type of operation currently being flattened or unflattened. It is one of the constants defined in the `gxGraphicsOperationOpcode` enumeration. (Set by QuickDraw GX.)

currentOperand

The type of data (within the current object) being flattened or unflattened. It is one of the constants defined in one of the data opcode enumerations, such as the `gxShapeDataOpcode` enumeration or the `gxStyleDataOpcode` enumeration. (Set by QuickDraw GX.)

compressed

The type of compression applied to the current item. (Set by QuickDraw GX.)

General information about flattening shapes is found in the section “Flattening and Unflattening Shapes” beginning on page 2-39. The `GXFlattenShape` function is described on page 2-88. The `GXUnflattenShape` function is described on page 2-90. The QuickDraw GX stream format, including the opcodes it uses and the types of compression it supports, is described in the stream format chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

The Hit-Test Info Structure

The hit-test info structure is a structure in which both the `GXHitTestShape` and `GXHitTestPicture` functions return information. `GXHitTestShape` uses only the first three fields; `GXHitTestPicture` uses all seven fields.

```
struct gxHitTestInfo {
    gxShapePart    what;
    long          index;
    Fixed         distance;
    gxShape       which;
    gxShape       containerPicture;
    long          containerIndex;
    long          totalIndex;
};
```

Shape Objects

Field descriptions

what	The parts of the shape that were hit, if any. QuickDraw GX returns in this field a mask denoting all shape parts (out of those specified for the hit-test) that are within the tolerance of the hit-test from the hit point. Shape parts are defined in the <code>gxShapeParts</code> enumeration; the tolerance and the subset of shape parts to test for make up the hit-test parameters. All are described in the chapter “Transform Objects” in this book.
index	The index of the nearest point in the geometry to the hit point. Every point in a shape’s geometry has an index number (indexes start at 1).
distance	The distance in geometry units from the hit point to the closest point on the shape part that was hit. (If no part was hit, this value is undefined.) If more than one shape part was hit, this is the distance to the first shape part encountered that is within the tolerance of the hit point. The order in which shape parts are examined during hit-testing is defined by the <code>gxShapeParts</code> enumeration, described in the chapter “Transform Objects” in this book.
which	A reference to the specific shape that was hit. (Used only by <code>GXHitTestPicture</code> .)
containerPicture	A reference to the picture shape that immediately contains the specific shape that was hit. Note that this may be a picture shape contained at some level within the picture shape specified in the call to <code>GXHitTestPicture</code> . (Used only by <code>GXHitTestPicture</code> .)
containerIndex	The index number—within the immediately containing shape—of the specific shape that was hit. (Used only by <code>GXHitTestPicture</code> .)
totalIndex	The index number—within the picture shape specified in the call to <code>GXHitTestPicture</code> —of the specific shape that was hit. (Used only by <code>GXHitTestPicture</code> .)

The `GXHitTestShape` function is described on page 2-86. The `GXHitTestPicture` function is described in the picture shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*.

Functions

This section describes the QuickDraw GX functions you can use to

- n create and manipulate a shape object
- n manipulate the properties of a shape object, including converting a shape from one type to another
- n directly manipulate a shape’s geometry
- n flatten and unflatten a shape
- n draw and hit-test a shape

Note

Shape-related QuickDraw GX functions not described in this section are listed and cross-referenced in Table 2-6 on page 2-42. u

Creating and Manipulating Shape Objects

The functions described in this section allow you to work with shapes as objects in memory. With the functions in this section, you can

- n determine the default shape object
- n create and dispose of a shape object
- n find the size of a shape object in memory
- n copy, clone, and compare shape objects
- n cache a shape object

GXGetDefaultShape

You can use the `GXGetDefaultShape` function to obtain a reference to the default shape object for a particular shape type.

```
gxShape GXGetDefaultShape(gxShapeType aType);
```

`aType` A shape type that specifies which default shape object to return.

function result A reference to the default shape for the shape type specified by the `aType` parameter.

DESCRIPTION

Note that the return value of this function is a reference to the actual default shape object, not a copy of it. If you edit the shape returned by this function, you alter the actual default shape object that the system uses when creating new shape objects.

You can also alter a default shape object by using the `GXSetDefaultShape` function.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`illegal_type_for_shape` (debugging version)

SEE ALSO

Default shape objects are discussed in the section “Default Shapes” beginning on page 2-18.

The `GXSetDefaultShape` function is described in the next section.

To create a copy of a default shape object, use the `GXNewShape` function, described on page 2-54.

GXSetDefaultShape

You can use the `GXSetDefaultShape` function to replace the default shape object of a particular shape type.

```
void GXSetDefaultShape(gxShape target);
```

`target` A reference to the new default shape object.

DESCRIPTION

The `GXSetDefaultShape` function replaces an existing default shape with the shape specified by the `target` parameter. The shape type of the target shape determines which default shape is replaced. This function disposes of the old default shape and increments the owner count of the target shape.

You can use the `GXSetDefaultShape` function to replace the style, ink, or transform of one of the default shapes by specifying a target shape with a different style, ink, or transform than the old default shape. When QuickDraw GX creates new shapes of the target shape’s shape type, the new shape will have the same ink, style, and transform as the target shape.

ERRORS, WARNINGS, AND NOTICES**Errors**

```
out_of_memory
shape_is_nil
illegal_type_for_shape      (debugging version)
```

SEE ALSO

Default shape objects are discussed in the section “Default Shapes” beginning on page 2-18.

To create a copy of a default shape object, use the `GXNewShape` function, described in the next section.

GXNewShape

You can use the `GXNewShape` function to create a new shape of a specified shape type.

```
gxShape GXNewShape(gxShapeType aType);
```

`aType` The type of shape object to create.

function result A reference to a newly created copy of the default shape object of the type specified by the `aType` parameter.

DESCRIPTION

The `GXNewShape` function creates a copy of the default shape object of the type specified by the `aType` parameter and gives it an owner count of 1.

Although this function creates a copy of the default shape, it does not create a copy of the default shape's style, ink, or transform. The new shape returned by this function contains references to same style, ink, and transform as the default shape. You can change the style, ink, and transform of the shape by using the functions `GXSetShapeStyle`, `GXSetShapeInk`, and `GXSetShapeTransform`.

You can use this function by itself to create empty and full shapes. For other shape types, you can use this function to create a shape and then you can customize the shape's geometry by using additional functions, such as `GXSetShapeGeometry` or one of the shape-specific functions such as `GXSetPoint`, `GXSetLine`, `GXSetPathParts`, or `GXSetGlyphParts`.

SPECIAL CONSIDERATIONS

If no error occurs, the `GXNewShape` function creates a shape object; you are responsible for disposing of that object when you no longer need it.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`illegal_type_for_shape` (debugging version)

SEE ALSO

Shape types, including empty and full shapes, are described in the section "Shape Type" beginning on page 2-9.

Default shape objects are discussed in the section "Default Shapes" beginning on page 2-18. To examine a default shape, use the `GXGetDefaultShape` function, described on page 2-52. To replace a default shape, use the `GXSetDefaultShape` function, described on page 2-53.

Shape Objects

The `GXSetShapeStyle` function is described on page 2-70; the `GXSetShapeInk` function is described on page 2-71; the `GXSetShapeTransform` function is described on page 2-73.

The `GXSetShapeGeometry` function is described on page 2-67. Other geometry-setting functions are described in the shape-specific chapters of *Inside Macintosh: QuickDraw GX Graphics* and *Inside Macintosh: QuickDraw GX Typography*.

For an example of the use of this function, see page 2-24.

GXDisposeShape

You can use the `GXDisposeShape` function to release a reference to a shape.

```
void GXDisposeShape(gxShape target);
```

`target` A reference to the shape to dispose of.

DESCRIPTION

The `GXDisposeShape` function decrements the owner count of the shape specified by the `target` parameter and releases any memory used by the shape if the owner count goes to 0.

SPECIAL CONSIDERATIONS

You cannot dispose of a shape that is locked, either because the `gxLockedShape` attribute is set or because `GXLockShape` was called to lock the shape. Depending on how the shape became locked, you must call `GXSetShapeAttributes` or `GXUnlockShape` before calling `GXDisposeShape` on a locked shape.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`
`shape_access_not_allowed` (debugging version)

Warnings

`cannot_dispose_default_shape` (debugging version)

SEE ALSO

Owner counts are discussed in the section “Copying, Comparing, and Cloning Shape Objects” beginning on page 2-25, and in the section “Manipulating a Shape Object’s Owner Count” beginning on page 2-31.

To examine the owner count of a shape, use the `GXGetShapeOwners` function, described on page 2-76. To increment the owner count of a shape, use the `GXCloneShape` function, described on page 2-61.

For an example of the use of this function, see page 2-24.

GXGetShapeSize

You can use the `GXGetShapeSize` function to determine the amount of memory currently occupied by a shape object.

```
long GXGetShapeSize(gxShape source);
```

source A reference to the shape object to determine the current memory size of.

function result The number of bytes of memory currently occupied by the shape specified in the *source* parameter.

DESCRIPTION

The `GXGetShapeSize` function takes the source shape’s type, owner count, fill, attributes, and geometry into consideration. It does not include the memory used by the shape’s style, ink, transform, or tag objects, but does include the memory used by the references to them.

The function result also includes the size of some shape properties private to QuickDraw GX, but does not include the size of the shape cache or the size of any memory overhead used to represent the shape.

This function returns only the memory size currently used by the shape. For example, when a shape is unloaded to disk it uses less memory, and the result of this function reflects its smaller size.

You can use the `GXLoadShape` function to load a shape into memory before determining its size.

ERRORS, WARNINGS, AND NOTICES**Errors**

`shape_is_nil`

SEE ALSO

To find the size of a shape's cache, use the `GXGetShapeCacheSize` function, described on page 2-64.

The `GXLoadShape` function is described in the memory management chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

For information about the memory size of graphic shapes, see the geometric shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*. For information about the memory size of typographic shapes, see the typographic shapes chapter of *Inside Macintosh: QuickDraw GX Typography*.

GXCopyToShape

The `GXCopyToShape` function copies the contents of one existing shape to another or else it creates a new shape and copies the contents of an existing shape to it. You can, for example, use this function to create a copy of a shape so that you can modify it without changing the original shape.

```
gxShape GXCopyToShape(gxShape target, gxShape source);
```

`target` A reference to the shape to copy the source shape's contents into. If you specify `nil` for this parameter, the function creates a new shape.

`source` A reference to the shape to copy from.

function result A reference to the copy (that is, the target shape).

DESCRIPTION

The `GXCopyToShape` function copies the properties and the geometry of the shape specified by the `source` parameter into the shape specified by the `target` parameter. It also copies the references to the source shape's ink, style, transform, and tags; that is, after the function returns, the target shape and the source shape share the same ink, style, transform, and tag objects. This function increments by 1 the owner counts of the source shape's ink, style, transform, and tag objects, and disposes of the original ink, style, transform, and tags of the target shape.

If you specify `nil` for the `target` parameter, this function creates a new shape to copy the contents of the source shape into.

SPECIAL CONSIDERATIONS

If you specify `nil` for the `target` parameter and no error occurs, the `GXCopyToShape` function creates a new shape object; you are responsible for disposing of that object when you no longer need it.

Shape Objects

If the target shape is locked, the `GXCopyToShape` function posts a `shape_access_not_allowed` error. If you try to copy a picture into a shape that is contained in the picture, this function posts a `picture_cannot_contain_itself` error. If you try to copy a shape of one type into the default shape of another type, this function posts a `cannot_dispose_default_shape` warning.

This function does not copy the pixel image of bitmap shapes, the shapes contained within picture shapes, or the set of style objects associated with glyph or layout shapes; instead, it copies the references to them. To obtain a complete copy of a bitmap shape, picture shape, glyph shape, or layout shape, use the `GXCopyDeepToShape` function.

ERRORS, WARNINGS, AND NOTICES

Errors

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>shape_access_not_allowed</code>	(debugging version)
<code>picture_cannot_contain_itself</code>	(debugging version)

Warnings

<code>cannot_dispose_default_shape</code>	(debugging version)
---	---------------------

SEE ALSO

To create a new shape that is a copy of the default shape instead of a copy of an existing shape, use the `GXNewShape` function, described on page 2-54.

The `GXCopyDeepToShape` function copies the pixel image of bitmap shapes and the shapes contained within picture shape; it is described in the next section. For more information about copying bitmap shapes and picture shapes, see *Inside Macintosh: QuickDraw GX Graphics*.

For information about copying typographic shapes, see the typographic shapes chapter of *Inside Macintosh: QuickDraw GX Typography*.

GXCopyDeepToShape

The `GXCopyDeepToShape` function copies the contents of one existing shape to another, or creates a new shape and copies the contents of an existing shape to it. For bitmap shapes, picture shapes, glyph shapes, and layout shapes, `GXCopyDeepToShape` copies more information than the `GXCopyToShape` function does.

```
gxShape GXCopyDeepToShape(gxShape target, gxShape source);
```

`target` A reference to the shape to copy the source shape's contents to. If you specify `nil` for this parameter, this function creates a new shape.

`source` A reference to the shape to copy from.

function result A reference to the copy (that is, the target shape).

DESCRIPTION

The `GXCopyDeepToShape` function copies the properties and the geometry of the shape specified by the `source` parameter into the shape specified by the `target` parameter. It also copies the references to the source shape's ink, style, transform, and tags; that is, after the function returns, the target shape and the source shape share the same ink, style, and transform objects. This function increments by 1 the owner counts of the source shape's ink, style, and transform, and disposes of the ink, style, and transform of the target shape.

If you specify `nil` for the `target` parameter, this function creates a new shape to copy the contents of the source shape into.

The `GXCopyDeepToShape` function is similar to the `GXCopyToShape` function except that it performs these additional operations:

- n For bitmap shapes, `GXCopyDeepToShape` also copies the complete pixel image.
- n For picture shapes, `GXCopyDeepToShape` also copies each shape in the source picture. If the source picture contains other picture shapes, their shapes are also recursively copied. The styles, inks, and transforms of the shapes within the picture are not copied; instead the copied shapes share references to the styles, inks, and transforms of the original shapes, and the `GXCopyDeepToShape` function increments by 1 the owner counts of the original styles, inks, and transforms.
- n For glyph and layout shapes, `GXCopyDeepToShape` also copies the set of style objects referenced in the style list that is part of the shape's geometry.

Because the `GXCopyDeepToShape` function copies the pixel image of bitmap shapes and the shapes contained within picture shapes, you can use it to create a copy of a bitmap or a picture, and then modify the copy without changing the original bitmap or picture.

SPECIAL CONSIDERATIONS

If you specify `nil` for the `target` parameter and no error occurs, the `GXCopyDeepToShape` function creates a new shape object; you are responsible for disposing of that object when you no longer need it.

If you try to copy a picture into a shape that is contained in the picture, the `GXCopyDeepToShape` function posts a `picture_cannot_contain_itself` error. If the target shape is locked, this function posts a `shape_access_not_allowed` error. If you try to copy a shape of one type into the default shape of another type, this function posts a `cannot_dispose_default_shape` warning.

ERRORS, WARNINGS, AND NOTICES**Errors**

out_of_memory
 shape_is_nil
 shape_access_not_allowed (debugging version)
 picture_cannot_contain_itself (debugging version)

Warnings

cannot_dispose_default_shape (debugging version)

SEE ALSO

To create a new shape that is a copy of the default shape instead of a copy of an existing shape, use the `GXNewShape` function, described on page 2-54.

To make a copy of an existing shape without copying all information for bitmap shapes, picture shapes, glyph shape, and layout shapes, use the `GXCopyToShape` function, described in the previous section.

For information about copying typographic shapes, see the typographic shapes chapter of *Inside Macintosh: QuickDraw GX Typography*.

GXEqualShape

You can use the `GXEqualShape` function to determine if two shapes are equal.

```
boolean GXEqualShape(gxShape one, gxShape two);
```

`one` A reference to one of the shapes to test for equality.

`two` A reference to the other shape to test for equality.

function result `true` if the shape specified by the `one` parameter is equal to the shape specified by the `two` parameter; `false` otherwise.

DESCRIPTION

The `GXEqualShape` function returns as its function result a Boolean value indicating whether the two QuickDraw GX shapes are equal. For two QuickDraw GX shapes to be equal, they must satisfy these requirements:

- n They must have the same shape type and fill, but they do not need to have the same attributes, owner count, or tag list.
- n Their geometries must have identical values; geometries that are equivalent but not identical are not considered to be equal. To eliminate false rejection of equivalent geometries, call the `GXSimplifyShape` function to simplify both shapes before you call `GXEqualShape`.

ERRORS, WARNINGS, AND NOTICES**Errors**

out_of_memory
shape_is_nil

SEE ALSO

Equivalent geometries are described in the section “Copying, Comparing, and Cloning Shape Objects” beginning on page 2-25. The `GXSimplifyShape` function is described in the geometric shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*.

To make a copy of a shape object that is equal by the criteria of this function, use the `GXCopyToShape` function, described on page 2-57.

GXCloneShape

You can use the `GXCloneShape` function to clone a shape—that is, to add a reference to it and increment its owner count.

```
gxShape GXCloneShape(gxShape source);
```

`source` A reference to the shape to clone.

function result A reference to the cloned shape.

DESCRIPTION

The `GXCloneShape` function returns a reference to the shape object specified by the `source` parameter and increments its owner count by 1. You typically use this function when you want to create another reference to an existing shape rather than create a distinct copy of the shape.

This function returns as its function result a reference to the shape—the same reference you pass in as the `source` parameter. Thus you can clone a shape with the following line of C code:

```
aShapeClone = GXCloneShape(aShape);
```

This line of code has almost the same affect as

```
aShapeClone = aShape;
```

that is, it sets the `aShapeClone` variable to reference the same shape object as the `aShape` variable. The difference is that `GXCloneShape` also increments the shape’s owner count.

ERRORS, WARNINGS, AND NOTICES**Errors**

`shape_is_nil`

SEE ALSO

Owner counts are discussed in the section “Copying, Comparing, and Cloning Shape Objects” beginning on page 2-25, and in the section “Manipulating a Shape Object’s Owner Count” beginning on page 2-31.

To examine the owner count of a shape, use the `GXGetShapeOwners` function, described on page 2-76. To decrement the owner count of a shape, use the `GXDisposeShape` function, described on page 2-55.

GXCacheShape

You can use the `GXCacheShape` function to prepare a shape for faster drawing.

```
void GXCacheShape(gxShape source);
```

`source` A reference to the shape to build the cache for.

DESCRIPTION

The `GXCacheShape` function prepares a shape for drawing by performing the calculations necessary to draw the shape and storing them in a shape cache. Then, when you draw the shape, time is saved because those calculations have already been made.

Although you do not need to call this function before drawing, you can use it to improve the speed of drawing on the screen.

To build a shape cache, use this function. To delete a shape cache, use the `GXDisposeShapeCache` function. To determine the amount of memory occupied by a shape cache, use the `GXGetShapeCacheSize` function.

SPECIAL CONSIDERATIONS

If you set the `gxCachedShape` attribute for a shape, QuickDraw GX automatically creates a cache and a compressed offscreen bitmap for the shape the first time it draws the shape. Unlike calling `GXCacheShape`, setting the `gxCachedShape` attribute can result in increased memory requirements for a shape.

ERRORS, WARNINGS, AND NOTICES

Because it performs preliminary calculations involved in drawing, the `GXCacheShape` function can, in addition to the errors listed below, post any errors and warnings associated with the `GXDrawShape` function. Therefore, `GXCacheShape` can post font-related errors if it is caching text.

Errors

```
out_of_memory
shape_is_nil
number_of_contours_exceeds_implementation_limit
number_of_points_exceeds_implementation_limit
size_of_polygon_exceeds_implementation_limit
size_of_path_exceeds_implementation_limit
size_of_bitmap_exceeds_implementation_limit
pattern_lattice_out_of_range (debugging version)
```

Warnings

```
character_substitution_took_place (debugging version)
graphic_type_cannot_be_dashed (debugging version)
unable_to_traverse_open_contour_that_starts_or_ends_off_the_curve (debugging version)
unable_to_draw_open_contour_that_starts_or_ends_off_the_curve (debugging version)
face_override_style_font_must_match_style (debugging version)
```

SEE ALSO

Shape caches are discussed in the section “Caching Shape Objects” beginning on page 2-27. The `gxCachedShape` attribute is described in that section and also in Table 2-4 on page 2-16.

The `GXGetShapeCacheSize` function is described on page 2-64. The `GXDisposeShapeCache` function is described in the next section.

For information about the caching and drawing typographic shapes, see the typographic shapes chapter of *Inside Macintosh: QuickDraw GX Typography*.

GXDisposeShapeCache

You can use the `GXDisposeShapeCache` function to release the memory occupied by a shape’s cache.

```
void GXDisposeShapeCache(gxShape target);
```

`target` A reference to the shape whose cache is to be disposed of.

DESCRIPTION

The `GXDisposeShapeCache` function immediately releases the memory allocated to the cache of the shape indicated by the `target` parameter. This function releases only that memory allocated to the target shape's cache. It does not release memory allocated to any related system caches or globals.

To build a shape cache, use the `GXCacheShape` function. To delete a shape cache, use this function. To determine the amount of memory occupied by a shape cache, use the `GXGetShapeCacheSize` function.

SPECIAL CONSIDERATIONS

You never need to call this function. QuickDraw GX disposes of caches automatically when it needs additional memory.

ERRORS, WARNINGS, AND NOTICES**Errors**

`shape_is_nil`

SEE ALSO

Shape caches are discussed in the section “Caching Shape Objects” beginning on page 2-27. The `GXCacheShape` function is described on page 2-62. The `GXGetShapeCacheSize` function is described in the next section.

GXGetShapeCacheSize

You can use the `GXGetShapeCacheSize` function to determine how much memory is allocated to a shape and its cache.

```
long GXGetShapeCacheSize(gxShape source);
```

`source` A reference to the shape object whose size in memory (including cache) is to be determined.

function result The approximate number of bytes of memory currently occupied by the shape and cache referenced in the `source` parameter.

DESCRIPTION

The `GXGetShapeCacheSize` function, like the `GXGetShapeSize` function, calculates the size of the source shape in memory, and does not include the memory used by the shape's referenced tags, style, ink, or transform. However, unlike `GXGetShapeSize`, this function result also includes the size of the source shape's current cache.

Shape Objects

This function returns only the memory size currently being used by the shape and its cache. If the shape is unloaded to disk, the result of this function indicates the smaller amount of memory used. If the shape has no cache, the result of this function is simply the memory size of the shape. You can use the `GXLoadShape` function to load a shape into memory before calling this function, to get the full size of the shape and cache.

In the interest of speed, this function provides only an approximation of the memory requirements of the shape's cache. The actual memory requirements of the cache depend on many factors, such as memory overhead, and would be less efficient to calculate. You can use this function to determine an approximate size for the memory partition needed for a set of shapes to be loaded and cached at the same time.

To determine the amount of memory occupied by a shape and its cache, use this function. To build a shape cache, use the `GXCacheShape` function. To delete a shape cache, use the `GXDisposeShapeCache` function.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`

SEE ALSO

Shape caches are discussed in the section “Caching Shape Objects” beginning on page 2-27. The `GXCacheShape` function is described on page 2-62. The `GXDisposeShapeCache` function is described in the previous section.

The `GXLoadShape` function is described in the memory management chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

To determine the amount of memory occupied by a shape without its cache, use the `GXGetShapeSize` function, described on page 2-56.

Manipulating Shape Object Properties

This section describes the functions available for manipulating the properties of shape objects. The functions described in this section allow you to

- n determine a shape object's type, geometry, and fill
- n determine the style, ink, and transform objects associated with a shape
- n determine a shape object's attributes
- n reset certain shape properties to their default values
- n find the owner count of a shape object
- n determine the tag objects associated with a shape object

Functions for direct manipulation of the geometry property of a shape object are described in the next section, “Directly Manipulating a Shape's Geometry” beginning on page 2-80.

GXGetShapeType

You can use the `GXGetShapeType` function to determine the shape type of a shape object.

```
gxShapeType GXGetShapeType(gxShape source);
```

`source` A reference to the shape object to determine the shape type of.

function result The shape type of the source shape.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`

SEE ALSO

Shape types are described in the section “Shape Type” beginning on page 2-9.

To assign a shape type to a shape object, use the `GXSetShapeType` function, described in the next section.

GXSetShapeType

You can use the `GXSetShapeType` function to convert a shape object from one shape type to another.

```
void GXSetShapeType(gxShape target, gxShapeType newType);
```

`target` A reference to the shape object to assign the new shape type to.

`newType` A reference to the shape type to be assigned to the shape.

DESCRIPTION

The `GXSetShapeType` function changes the type of the target shape to the shape type specified by `newType`. Many different kinds of conversions are possible: typographic types can be converted to other typographic types or to graphic types; graphic types can be converted to other graphic types. The results of the conversion differ in each case, depending on which type is converted to which other type. See Table 2-5 on page 2-33 for a list of chapters that describe how conversion works for different shape types.

ERRORS, WARNINGS, AND NOTICES

When you change a shape to a bitmap type, the `GXSetShapeType` function performs preliminary calculations on its data and thus may post, in addition to the errors listed below, errors associated with the `GXDrawShape` function. When you change a shape to a typographic type, `GXSetShapeType` may post font-related errors.

Errors

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>illegal_type_for_shape</code>	(debugging version)
<code>shape_access_not_allowed</code>	(debugging version)

Warnings

<code>new_shape_contains_invalid_data</code>	(debugging version)
--	---------------------

Notices (debugging version)

<code>shape_type_already_set</code>	
-------------------------------------	--

SEE ALSO

What happens when you call `GXSetShapeType` to convert shapes of one type to another is described in *Inside Macintosh: QuickDraw GX Graphics* and *Inside Macintosh: QuickDraw GX Typography*. Table 2-5 on page 2-33 shows which specific chapters to read for detailed information on conversion among the various types of shapes.

Shape types are described in the section “Shape Type” beginning on page 2-9 of this chapter.

To determine the shape type of a shape object, use the `GXGetShapeType` function, described in the previous section.

GXSetShapeGeometry

You can use the `GXSetShapeGeometry` function to copy the geometry from one shape object to another.

```
void GXSetShapeGeometry(gxShape target, gxShape geometry);
```

`target` A reference to the shape to copy the new geometry into.

`geometry` A reference to the shape to copy the new geometry from.

DESCRIPTION

For two shape objects with the same shape type, the `GXSetShapeGeometry` function copies the geometry from the shape referenced by the `geometry` parameter to the shape referenced by the `target` parameter. If the type of the shape referenced in the `geometry` parameter is different from the type of the target shape, the target shape becomes the geometry shape’s type.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`shape_is_nil`
`picture_cannot_contain_itself`
`shape_access_not_allowed` (debugging version)

SEE ALSO

To directly manipulate the contents of a shape's geometry, see the section "Directly Manipulating a Shape's Geometry" beginning on page 2-34; see also the descriptions of the `GXLockShape`, `GXUnlockShape`, `GXGetShapeStructure`, and `GXChangedShape` functions, beginning on page 2-80.

Specific methods for setting and manipulating the geometries of graphic shapes are described in *Inside Macintosh: QuickDraw GX Graphics*. Methods for setting and manipulating the geometries of typographic shapes are described in *Inside Macintosh: QuickDraw GX Typography*.

GXGetShapeFill

You can use the `GXGetShapeFill` function to retrieve the fill property of a shape object.

```
gxShapeFill GXGetShapeFill(gxShape source);
```

`source` A reference to the shape whose fill property you want to retrieve.

function result The fill of the source shape.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`shape_is_nil`

SEE ALSO

Shape fills are described in the section "Shape Fill" beginning on page 2-13.

To assign a fill to a shape object, use the `GXSetShapeFill` function, described in the next section.

For more information on shape fill as it applies to geometric shapes, see the geometric shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*. For more information on shape fill as it applies to typographic shapes, see the typographic shapes chapter of *Inside Macintosh: QuickDraw GX Typography*.

GXSetShapeFill

You can use the `GXSetShapeFill` function to change the fill property of a shape object.

```
void GXSetShapeFill(gxShape target, gxShapeFill newFill);
```

`target` A reference to the shape whose fill property you want to change.

`newFill` The new value for shape fill.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`

`shape_is_nil`

`shape_access_not_allowed` (debugging version)

`parameter_out_of_range` (debugging version)

`inconsistent_parameters` (debugging version)

Notices (debugging version)

`fill_already_set`

SEE ALSO

Shape fills are described in the section “Shape Fill” beginning on page 2-13.

This function is further described for geometric shapes in the geometric shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*, and for typographic shapes in the typographic shapes chapter of *Inside Macintosh: QuickDraw GX Typography*.

To determine the shape fill of a shape object, use the `GXGetShapeFill` function, described in the previous section.

GXGetShapeStyle

You can use the `GXGetShapeStyle` function to determine the style object associated with a QuickDraw GX shape.

```
gxStyle GXGetShapeStyle(gxShape source);
```

`source` A reference to the shape object whose style object is to be determined.

function result A reference to the style object associated with the source shape object.

ERRORS, WARNINGS, AND NOTICES**Errors**

out_of_memory
shape_is_nil

SEE ALSO

The relationship of style objects to QuickDraw GX shapes is discussed in “About QuickDraw GX Shapes” beginning on page 2-5. Style objects themselves are further discussed in the chapter “Style Objects” in this book.

To change the style object associated with a QuickDraw GX shape, use the `GXSetShapeStyle` function, described in the next section.

GXSetShapeStyle

You can use the `GXSetShapeStyle` function to change the style object associated with a QuickDraw GX shape.

```
void GXSetShapeStyle(gxShape target, gxStyle newStyle);
```

`target` A reference to the shape whose style object is to be changed.
`newStyle` A reference to the new style object to associate with the `target` shape.

DESCRIPTION

The `GXSetShapeStyle` function disassociates the style object already associated with the `target` shape and disposes of it. The function then assigns the style object referenced by the `newStyle` parameter to the target shape and increments by 1 the owner count of the new style object.

ERRORS, WARNINGS, AND NOTICES**Errors**

out_of_memory
shape_is_nil
style_is_nil
shape_access_not_allowed (debugging version)

Notices (debugging version)

style_already_set

SEE ALSO

The relationship of style objects to QuickDraw GX shapes is discussed in “About QuickDraw GX Shapes” beginning on page 2-5. Style objects themselves are further discussed in the chapter “Style Objects” in this book.

To determine the style object associated with a QuickDraw GX shape, use the `GXGetShapeStyle` function, described in the previous section.

GXGetShapeInk

You can use the `GXGetShapeInk` function to determine the ink object associated with a shape object.

```
gxInk GXGetShapeInk(gxShape source);
```

`source` A reference to the shape whose ink object is to be determined.

function result A reference to the ink object associated with the source shape object.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`shape_is_nil`

SEE ALSO

The relationship of ink objects to QuickDraw GX shapes is discussed in “About QuickDraw GX Shapes” beginning on page 2-5. Ink objects themselves are further discussed in the chapter “Ink Objects” in this book.

To change the ink object associated with a QuickDraw GX shape, use the `GXSetShapeInk` function, described in the next section.

GXSetShapeInk

You can use the `GXSetShapeInk` function to change the ink object associated with a shape object.

```
void GXSetShapeInk(gxShape target, gxInk newInk);
```

`target` A reference to the shape whose ink object is to be changed.

`newInk` A reference to the new ink object to associate with the target shape.

DESCRIPTION

The `GXSetShapeInk` function disassociates the ink object already associated with the target shape and disposes of it. The function then assigns the ink object referenced by the `newInk` parameter to the target shape and increments the owner count of the new ink object.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`shape_is_nil`
`ink_is_nil`
`shape_access_not_allowed` (debugging version)

Notices (debugging version)

`ink_already_set`

SEE ALSO

The relationship of ink objects to QuickDraw GX shapes is discussed in “About QuickDraw GX Shapes” beginning on page 2-5. Ink objects themselves are further discussed in the chapter “Ink Objects” in this book.

To determine the ink object associated with a QuickDraw GX shape, use the `GXGetShapeInk` function, described in the previous section.

GXGetShapeTransform

You can use the `GXGetShapeTransform` function to determine the transform object associated with a shape object.

```
gxTransform GXGetShapeTransform(gxShape source);
```

source A reference to the shape whose transform object is to be determined.

function result A reference to the transform object associated with the source shape object.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`shape_is_nil`

SEE ALSO

The relationship of transform objects to QuickDraw GX shapes is discussed in “About QuickDraw GX Shapes” beginning on page 2-5. Transform objects themselves are further discussed in the chapter “Transform Objects” in this book.

To change the transform object associated with a QuickDraw GX shape, use the `GXSetShapeTransform` function, described in the next section.

GXSetShapeTransform

You can use the `GXSetShapeTransform` function to change the transform object associated with a shape object.

```
void GXSetShapeTransform(gxShape target,
                        gxTransform newTransform);
```

`target` A reference to the shape whose transform object is to be changed.

`newTransform` A reference to the new transform object to associate with the target shape.

DESCRIPTION

The `GXSetShapeTransform` function disassociates the transform object already associated with the target shape and disposes of it. `GXSetShapeTransform` then assigns the transform object referenced by the `newTransform` parameter to the target shape and increments by 1 the owner count of the new transform object.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`shape_is_nil`
`transform_is_nil`
`shape_access_not_allowed` (debugging version)

Notices (debugging version)

`transform_already_set`

SEE ALSO

The relationship of transform objects to QuickDraw GX shapes is discussed in “About QuickDraw GX Shapes” beginning on page 2-5. Transform objects themselves are further discussed in the chapter “Transform Objects” in this book.

To determine the transform object associated with a QuickDraw GX shape, use the `GXGetShapeTransform` function, described in the previous section.

GXGetShapeAttributes

You can use the `GXGetShapeAttributes` function to examine which attributes of a shape object are set.

```
gxShapeAttribute GXGetShapeAttributes(gxShape source);
```

`source` A reference to the shape to find the attributes of.

function result The shape attributes of the source shape.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`shape_is_nil`

SEE ALSO

Shape attributes are described in the section “Shape Attributes” beginning on page 2-16, and in the section “Getting and Setting a Shape Object’s Type, Fill, and Attributes” beginning on page 2-28.

To change the attributes of a shape object, use the `GXSetShapeAttributes` function, described in the next section.

For an example of the use of this function, see page 2-29.

GXSetShapeAttributes

You can use the `GXSetShapeAttributes` function to set or clear the attributes for a particular shape object.

```
void GXSetShapeAttributes(gxShape target, gxShapeAttribute  
attributes);
```

`target` A reference to the shape object to change the attributes of.

`attributes` The new shape attributes to be assigned.

ERRORS, WARNINGS, AND NOTICES**Errors**

out_of_memory	
shape_is_nil	
parameter_out_of_range	(debugging version)
inconsistent_parameters	(debugging version)
shape_access_not_allowed	(debugging version)

Warnings

picture_expected	
cannot_set_unique_items_attribute_when_picture_contains_items	

SEE ALSO

Shape attributes are described in the section “Shape Attributes” beginning on page 2-16, and in the section “Getting and Setting a Shape Object’s Type, Fill, and Attributes” beginning on page 2-28.

To examine the attributes of a shape object, use the `GXGetShapeAttributes` function, described in the previous section.

For an example of the use of this function, see page 2-29.

GXResetShape

You can use the `GXResetShape` function to reset the attributes, fill, style, ink, and transform of a shape to their default values.

```
void GXResetShape(gxShape target);
```

`target` A reference to the shape object whose properties you want to reset.

DESCRIPTION

The `GXResetShape` function resets the shape attributes and the shape fill of the shape object specified by the `target` parameter to match the shape attributes and shape fill of the corresponding default shape. The function also resets the style, ink, and transform references of the target shape to their default values.

The `GXResetShape` function does not change the target shape’s geometry, owner count, or tags.

After the `GXResetShape` function returns, the target shape references the same style, ink, and transform as the corresponding default shape object. The `GXResetShape` function increments by 1 the owner counts of the default style, ink, and transform, and disposes of the target shape’s original style, ink, and transform.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`shape_is_nil`
`shape_access_not_allowed` (debugging version)

SEE ALSO

Default shape objects are described in the section “Default Shapes” beginning on page 2-18.

To examine a default shape, use the `GXGetDefaultShape` function, described on page 2-52. To replace a default shape, use the `GXSetDefaultShape` function, described on page 2-53.

For information on resetting typographic shapes, see the typographic shapes chapter of *Inside Macintosh: QuickDraw GX Typography*.

GXGetShapeOwners

You can use the `GXGetShapeOwners` function to determine the number of references to a particular shape object.

```
long GXGetShapeOwners(gxShape source);
```

`source` A reference to the shape to find the owner count of.

function result The owner count of the source shape.

DESCRIPTION

The `GXGetShapeOwners` function returns as its function result the current number of references to the shape object.

ERRORS, WARNINGS, AND NOTICES**Errors**

`shape_is_nil`

SEE ALSO

Owner counts for shape objects are discussed in the section “Copying, Comparing, and Cloning Shape Objects” beginning on page 2-25, and in the section “Manipulating a Shape Object’s Owner Count” beginning on page 2-31.

To increment the owner count of a shape, use the `GXCloneShape` function, described on page 2-61. To decrement the owner count of a shape, use the `GXDisposeShape` function, described on page 2-55.

GXGetShapeTags

You can use the `GXGetShapeTags` function to examine one or more of the tag objects associated with a shape object.

```
long GXGetShapeTags(gxShape source, long tagType, long index,
                   long count, gxTag items[]);
```

<code>source</code>	A reference to the shape object to examine the tag list of.
<code>tagType</code>	The type of tag object to search for. A value of 0 indicates that you want to look for all tag types.
<code>index</code>	The (1-based) index of the first such tag reference to return.
<code>count</code>	The number of tag references to return.
<code>items</code>	An array to hold the returned tag references.

function result The number of tag references found that fit the criteria.

DESCRIPTION

The `GXGetShapeTags` function searches the tag list of the `source` shape object for references to tag objects with the tag type specified by the `tagType` parameter. If you specify 0 for the `tagType` parameter, the `GXGetShapeTags` function searches all tag types.

You can use the `index` and `count` parameters to specify which tag references of the appropriate type the `GXGetShapeTags` function should return. The `index` parameter indicates the first tag reference to return and the `count` parameter indicates how many tag references to return. The `index` parameter must be greater than 0. The `count` parameter must be greater than 0 or equal to the `gxSelectToEnd` constant (-1), which indicates that all tag references (starting with the tag reference indicated by the `index` parameter) should be returned.

If you pass a value other than `nil` for the `items` parameter, the `GXGetShapeTags` function returns in it the tag references that were found.

ERRORS, WARNINGS, AND NOTICES**Errors**

```

out_of_memory
shape_is_nil
index_is_less_than_one      (debugging version)
count_is_less_than_one     (debugging version)

```

Warnings

```

index_out_of_range
count_out_of_range

```

SEE ALSO

Tag objects are introduced in the chapter “Introduction to Objects” in this book. Functions to create and manipulate tags objects, and a list of reserved tag types, are described in the chapter “Tag Objects” in this book.

To change the set of tag references associated with a shape, use the `GXSetShapeTags` function, described next.

GXSetShapeTags

You can use the `GXSetShapeTags` function to add, remove, or replace tag objects associated with a shape object.

```

void GXSetShapeTags(gxShape target, long tagType, long index,
                   long oldCount, long newCount,
                   const gxTag items[]);

```

<code>target</code>	A reference to the shape object to alter the tag list of.
<code>tagType</code>	The type of tag objects to replace. A value of 0 indicates that you want to replace tags of all types.
<code>index</code>	The (1-based) index of the first tag reference (to a tag object of the appropriate type) to replace.
<code>oldCount</code>	The number of tag references to replace. A value of 0 specifies that you want to insert tag references before the tag reference indicated by the <code>index</code> parameter, rather than replace tag references. A value of -1 (the <code>gxSelectToEnd</code> constant) specifies that all tag references of the requested type, starting with the tag reference indicated by the <code>index</code> parameter, should be replaced.
<code>newCount</code>	The number of tag references to insert. A value of 0 specifies that there are no tag references to insert; the existing tag references that match the criteria you specify are removed from the source shape’s tag list and disposed of.
<code>items</code>	An array of tag references to insert in the tag list.

DESCRIPTION

The `GXSetShapeTags` function allows you to add tag references to a shape object's tag list, to remove tag references from the list, or to replace tag references in the list with new tag references. In any of these three cases, the `target` parameter specifies the shape object to be modified, the `newCount` parameter specifies the number of tag references to add, and the `items` parameter provides the new tag references.

- n To add tag references, set the `oldCount` parameter to 0. Use the `tagType` and the `index` parameters to specify where to add the new tag references. (For example, if you specify `nil` for the `tagType` parameter and 1 for the `index` parameter, this function inserts the new tag references before the current tag references. If you specify a value other than `nil` for the `tagType` parameter and a value of 2 for the `index` parameter, the function inserts the new tag references before the second tag reference with a tag type matching the `tagType` parameter.)
- n To remove tag references, set the `newCount` parameter to 0 and the `items` parameter to `nil`. You can use the `index` and the `oldCount` parameters to specify which tag references (of the specified type) should be removed. The `index` parameter indicates the first tag reference (of the specified type) to remove and the `oldCount` parameter indicates how many tag references (of the specified type) to remove.
- n To replace tag references, use the `tagType`, `index`, and `oldCount` parameters to indicate which tag references to replace, and use the `newCount` and `items` parameters to specify the new tag references to add. If `newCount` is greater than `oldCount`, the extra tag references are placed immediately adjacent to the last tag reference replaced.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>tag_is_nil</code>	
<code>parameter_is_nil</code>	(debugging version)
<code>inconsistent_parameters</code>	(debugging version)
<code>parameter_out_of_range</code>	(debugging version)
<code>index_is_less_than_zero</code>	(debugging version)
<code>cannot_dispose_locked_tag</code>	(debugging version)

Warnings

`index_out_of_range`
`count_out_of_range`

Notices (debugging version)

`tag_already_set`

SEE ALSO

Tag objects are introduced in the chapter “Introduction to Objects” in this book. Functions to create and manipulate tags objects, and a list of reserved tag types, are described in the chapter “Tag Objects” in this book.

To examine the set of tag references associated with a shape, use the `GXGetShapeTags` function, described in the previous section.

Directly Manipulating a Shape's Geometry

This section describes the functions you use to directly manipulate the geometry of a shape object. Unlike most calls to QuickDraw GX objects, these functions give you direct access to the data of a geometry—in QuickDraw GX memory—without regard to the shape object it is part of. You typically call the functions in this order:

- n GXLockShape
- n GXGetShapeStructure
- n GXUnlockShape
- n GXChangedShape

GXLockShape

You can use the `GXLockShape` function to load a shape into memory and lock its geometry into a fixed memory location.

```
void GXLockShape(gxShape target);
```

`target` A reference to the shape to be loaded and locked.

DESCRIPTION

The `GXLockShape` function prevents a shape from being relocated. You must set the `gxDirectShape` attribute of the target shape before calling this function.

To avoid fragmenting the QuickDraw GX heap, call the `GXUnlockShape` function as soon as possible after calling `GXLockShape`.

To directly edit a shape's geometry, call `GXLockShape` followed by `GXGetShapeStructure`. After editing, call `GXUnlockShape` followed by `GXChangedShape`.

SPECIAL CONSIDERATIONS

The `GXLockShape` function is not related to the `gxLockedShape` shape attribute. If you set the `gxLockedShape` attribute, you cannot alter the shape's geometry with functions such as `GXSetPoint` and `GXSetRectangle`, described in the geometric shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*. Setting `gxLockedShape` has no effect on the direct manipulation of geometry using the calls described here.

In low memory conditions with fragmented memory, QuickDraw GX can unlock locked objects and relocate them. Be careful about making memory-intensive calls after locking an object.

ERRORS, WARNINGS, AND NOTICES**Errors**

out_of_memory
 shape_is_nil
 graphic_type_does_not_have_a_structure (debugging version)

Notices (debugging version)

directShape_attribute_set_as_side_effect

SEE ALSO

The `GXUnlockShape`, `GXGetShapeStructure`, and `GXChangedShape` functions are described in the following three sections.

Shape attributes are described in the section “Shape Attributes” beginning on page 2-16. To set shape attributes, use the `GXSetShapeAttributes` function, described on page 2-74.

GXUnlockShape

You can use the `GXUnlockShape` function to allow QuickDraw GX to relocate, compress, or unload a shape that has been locked.

```
void GXUnlockShape(gxShape target);
```

target A reference to the shape to unlock.

DESCRIPTION

The `GXUnlockShape` function releases a previously locked shape for relocation or other movement.

To directly edit a shape’s geometry, call `GXLockShape` followed by `GXGetShapeStructure`. After editing, call `GXUnlockShape` followed by `GXChangedShape`. Once you call `GXUnlockShape`, the shape’s geometry may be relocated and a pointer returned by `GXGetShapeStructure` may no longer be valid.

SPECIAL CONSIDERATIONS

To avoid fragmenting the QuickDraw GX heap, call the `GXUnlockShape` function as soon as possible after calling `GXLockShape`.

ERRORS, WARNINGS, AND NOTICES**Errors**

shape_is_nil

Notices (debugging version)

shape_not_locked

SEE ALSO

The `GXLockShape` function is described in the previous section. The `GXGetShapeStructure` and `GXChangedShape` functions are described in the following two sections.

The `GXDisposeShape` function is described on page 2-55.

GXGetShapeStructure

You can use the `GXGetShapeStructure` function to get a pointer to the geometry of a shape object.

```
void *GXGetShapeStructure(gxShape source, long *length);
```

`source` A reference to the shape object whose geometry you need access to.

`length` A pointer to a `long` value. On return, the value specifies the size in bytes of the shape's geometry.

function result A pointer to the geometry of the source shape object.

DESCRIPTION

The `GXGetShapeStructure` function determines the size of a shape's geometry and returns a pointer to the geometry in the QuickDraw GX heap. You can use the pointer to examine or change the geometry without copying the geometry into your application's heap and back again.

Before calling `GXGetShapeStructure`, you should first call `GXLockShape` to prevent the geometry from being relocated and you should set the `gxDirectShape` attribute to make the shape accessible in the QuickDraw GX heap. After you are finished examining or changing the geometry, call `GXUnlockShape`. If you change the shape's geometry, you must call the `GXChangedShape` function to notify QuickDraw GX that the shape's cache is no longer valid.

To edit a geometry, you need to know its structure. `GXGetShapeStructure` returns a pointer and a size only; it does not provide you with any information about the internal structure of the geometry. For example, if the source shape is a path, you must cast the function result to a `gxPaths` pointer. Such information is not described in this book.

If you call this function for a shape that has no geometry (shape types `gxEmptyType` and `gxFullType`), the function posts a `graphic_type_has_no_structure` warning.

SPECIAL CONSIDERATIONS

If you do not set the `gxDirectShape` attribute or do not lock the shape, QuickDraw GX does them for you as a side effect of the `GXGetShapeStructure` function call. You must still call `GXUnlockShape` to unlock the shape and, if you wish, reset the attribute.

This function is rarely needed. In most instances, you can manipulate a shape's geometry with calls to geometry-specific functions such as `GXGetRectangle` or `GXGetGlyphTangents`. This function is provided as a fast alternative to those functions, but be aware that it may fail in low-memory conditions; see "Special Considerations" under the description of the `GXLockShape` function, on page 2-80.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`shape_is_nil`
`graphic_type_does_not_have_a_structure` (debugging version)

Notices (debugging version)

`lockShape_called_as_side_effect`

SEE ALSO

The `GXLockShape` and `GXUnlockShape` functions are described in the previous sections. The `GXChangedShape` function is described in the following section.

Shape types are described in the section "Shape Type" beginning on page 2-9.

Shape geometry structures, and the functions for manipulating them, are described in the shape-specific chapters of *Inside Macintosh: QuickDraw GX Graphics* and *Inside Macintosh: QuickDraw GX Typography*.

GXChangedShape

You can use the `GXChangedShape` function to notify QuickDraw GX that you have directly edited the geometry of a shape.

```
void GXChangedShape(gxShape target);
```

`target` A reference to the shape object whose geometry you have directly edited.

DESCRIPTION

The `GXChangedShape` function notifies QuickDraw GX that the geometry of the shape referenced by the `target` parameter has been modified. QuickDraw GX can then use that information to invalidate existing shape caches, if necessary.

You need to call this function only if you have directly edited a shape's geometry by using the pointer returned by the `GXGetShapeStructure` function. If you edit a shape geometry using any other shape-editing function, you do not need to call `GXChangedShape`.

To directly edit a shape's geometry, call `GXLockShape` followed by `GXGetShapeStructure`. After editing, call `GXUnlockShape` followed by `GXChangedShape`.

ERRORS, WARNINGS, AND NOTICES**Errors**

`shape_is_nil`

SEE ALSO

The `GXLockShape`, `GXUnlockShape`, and `GXGetShapeStructure` functions are described in the previous sections.

Shape caches are discussed in the section "Caching Shape Objects" beginning on page 2-27.

Other functions for editing shape geometries are described in the shape-specific chapters of *Inside Macintosh: QuickDraw GX Graphics* and *Inside Macintosh: QuickDraw GX Typography*.

Drawing and Hit-Testing Shapes

This section describes the basic QuickDraw GX functions for drawing and hit-testing shapes: `GXDrawShape` and `GXHitTestShape`.

GXDrawShape

You can use the `GXDrawShape` function to draw any shape.

```
void GXDrawShape(gxShape source);
```

`source` A reference to the shape object of the shape to draw.

DESCRIPTION

The `GXDrawShape` function draws the shape referenced by the `source` parameter, taking into account the properties specified in the shape's style, ink, and transform objects. It draws the shape to the display device or devices specified indirectly in the shape's transform object.

As part of preparation for drawing, QuickDraw GX makes preliminary calculations and stores the results in caches. You can in some cases speed drawing by having the calculations and cache storage occur ahead of time; you can do that by setting the source shape's `gxCachedShape` attribute or by calling the `GXCacheShape` function.

ERRORS, WARNINGS, AND NOTICES

In addition to the errors listed below, the `GXDrawShape` function can post font-related errors if it is drawing text.

Errors

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>number_of_contours_exceeds_implementation_limit</code>	
<code>number_of_points_exceeds_implementation_limit</code>	
<code>size_of_polygon_exceeds_implementation_limit</code>	
<code>size_of_path_exceeds_implementation_limit</code>	
<code>size_of_bitmap_exceeds_implementation_limit</code>	
<code>pattern_lattice_out_of_range</code>	(debugging version)

Warnings

<code>character_substitution_took_place</code>	
<code>graphic_type_cannot_be_dashed</code>	(debugging version)
<code>unable_to_traverse_open_contour_that_starts_or_ends_off_the_curve</code>	(debugging version)
<code>unable_to_draw_open_contour_that_starts_or_ends_off_the_curve</code>	(debugging version)
<code>face_override_style_font_must_match_style</code>	(debugging version)

SEE ALSO

The `GXDrawShape` function as applied to geometric shapes, and other functions for drawing geometric shapes, are described in the geometric shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*. The function as applied to typographic shapes, and other functions for drawing typographic shapes, are described in the text shapes, glyph shapes, and layout shapes chapters of *Inside Macintosh: QuickDraw GX Typography*.

Transform objects and their relation to display devices are described in the chapter "Transform Objects" in this book.

The `gxCachedShape` attribute is described in Table 2-4 on page 2-16. The `GXCacheShape` function is described on page 2-62. The differences between the two caching methods are described in the section "Caching Shape Objects" beginning on page 2-27.

GXHitTestShape

You can use the `GXHitTestShape` function to convert a point in space (which may represent, for example, the location of a mousedown event) into a distance from a particular part of the geometry of a shape.

```
gxShapePart GXHitTestShape(gxShape target, const gxPoint *test,
                           gxHitTestInfo *result);
```

<code>target</code>	A reference to the shape to hit-test.
<code>test</code>	A pointer to a point structure specifying the location to hit-test the shape against. The location must be specified in the local coordinates of the shape.
<code>result</code>	A pointer to a <code>gxHitTestInfo</code> structure. On return, the structure contains detailed information about the hit-test.

function result The parts of the shape corresponding to the location specified in the `test` parameter (within the tolerance limits for the hit-test).

DESCRIPTION

The `GXHitTestShape` function takes a shape reference and a point in geometry or local space and returns whether or not the point was within a certain distance (tolerance) of one of a set of specified parts of the shape. With this function you can, for example, respond to user actions such as mouse clicks or movements by highlighting or selecting parts of shapes. The tolerance and the shape parts are defined in the hit-test parameters of the shape's transform object. The function returns the shape parts that were hit, or else the value `gxNoPart` if no tested part of the shape was hit.

On return, the `result` parameter contains a filled-out `gxHitTestInfo` structure. Only the first three fields are filled out by `GXHitTestShape`:

- n The `what` field describes the shape parts that were hit, if any. It is identical to the function result from this function.
- n The `index` field identifies, by (1-based) index, the nearest point in the geometry to the hit point.
- n The `distance` field describes the distance from the hit point to the closest point on the shape part that was hit. (If no part was hit, this value is undefined.) If more than one shape part was hit, this is the distance to the first shape part encountered that was within the tolerance of the hit point. The order in which shape parts are examined during hit-testing is defined by the `gxShapeParts` enumeration.

ERRORS, WARNINGS, AND NOTICES

Because it performs many of the calculations involved in drawing, the `GXHitTestShape` function can post, in addition to the errors listed below, any errors and warnings associated with the `GXDrawShape` function. Therefore `GXHitTestShape` can post font-related errors if it is caching text.

Errors

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>parameter_is_nil</code>	(debugging version)
<code>number_of_contours_exceeds_implementation_limit</code>	
<code>number_of_points_exceeds_implementation_limit</code>	
<code>size_of_polygon_exceeds_implementation_limit</code>	
<code>size_of_path_exceeds_implementation_limit</code>	
<code>size_of_bitmap_exceeds_implementation_limit</code>	
<code>pattern_lattice_out_of_range</code>	(debugging version)

Warnings

<code>character_substitution_took_place</code>	(debugging version)
<code>graphic_type_cannot_be_dashed</code>	(debugging version)
<code>unable_to_traverse_open_contour_that_starts_or_ends_off_the_curve</code>	(debugging version)
<code>unable_to_draw_open_contour_that_starts_or_ends_off_the_curve</code>	(debugging version)
<code>face_override_style_font_must_match_style</code>	(debugging version)

SEE ALSO

Hit-testing is discussed in the section “Drawing and Hit-Testing Shapes” beginning on page 2-35.

The `gxHitTestInfo` structure is described on page 2-50.

The `gxShapeParts` enumeration and the `gxShapePart` mask are also described in the hit-test parameters section of the chapter “Transform Objects” in this book.

Flattening and Unflattening Shape Objects

The two functions described in this section allow you to convert shapes into a compressed, stream-based format for storage or transmission, and to reconstruct shapes from their compressed form.

GXFlattenShape

You can use the `GXFlattenShape` function to convert the object form of a shape—including all the objects that it references—into a stream format that is public and suitable for storage and parsing.

```
void GXFlattenShape(gxShape source, gxFlattenFlag flags,
                   struct gxSpoolBlock *block);
```

<code>source</code>	A reference to the shape you wish to flatten.
<code>flags</code>	A set of flags that specify whether or not to save additional information with the flattened file.
<code>block</code>	A pointer to a spool block structure. QuickDraw GX uses information in the spool block to create and store the flattened data.

DESCRIPTION

The `GXFlattenShape` function creates a flattened version of the shape referenced by the `source` parameter and places it into a buffer pointed to by the spool block specified in the `block` parameter.

Before calling `GXFlattenShape`, you need to allocate a spool block structure and a buffer to hold the flattened data, and place a pointer to the buffer and a specification of its size into the spool block. You also place into the spool block a pointer to an application-defined spool function that writes the flattened data from the buffer to a file. The spool function responds to commands from QuickDraw GX to open, write, and close the file used to hold the flattened data.

If your spool block structure specifies `nil` for the buffer pointer and 0 for its size, QuickDraw GX allocates a default buffer (512 KB in version 1.0 of QuickDraw GX) for you.

Upon completion of the function, QuickDraw GX writes into the spool block the number of bytes of flattened data it has placed into the buffer. It also writes other information into the spool block; your spool function can use that information if you want it to parse the flattened file as flattening occurs. Normally, however, for simple flattening of shapes, your application need not read any of the information returned in the spool block, and your spool function needs to read only the size of the flattened data in the buffer.

Note that flattening a shape causes flattened versions of all its referenced objects, such as its style, ink, and transform—and all of their referenced objects in turn—to be stored as well. To flatten a group of shapes, place them in a picture and flatten the picture.

If you set the `gxFontListFlatten`, `gxFontGlyphsFlatten`, or `gxFontVariationsFlatten` flag in the `flags` parameter when calling this function, `GXFlattenShape` creates a tag object and attaches it to the source shape. The tag object is of type `'flst'` and lists the names of the fonts referenced in the shape, the individual glyphs used in the shape, or the descriptions of any font variations used in the shape, respectively.

Shape Objects

If you set the `gxBitmapAliasFlatten` flag in the `flags` parameter when calling this function, `GXFlattenShape` includes with the flattened shape all image data from any bitmap shapes that are referenced by the shape. If this flag is not set, image data from bitmap shapes whose image data is disk-based is not included in the flattened shape. That image data is not lost, however, because a tag object specifying the file holding the image data is flattened along with the shape.

The flattened stream created by `GXFlattenShape` consists of a series of opcodes and associated data, following the QuickDraw GX stream format.

SPECIAL CONSIDERATIONS

If the source shape already has a tag object of type `'flst'` attached to it, `GXFlattenShape` replaces that tag with a new tag of type `'flst'`; it also posts a `tags_of_type_flst_removed` warning.

If the `block` parameter is `nil`, this function returns a `parameter_is_nil` error. If the `spool-function` pointer in the `spool` block passed in the `block` parameter is `nil`, this function returns a `spoolProcedure_is_nil` error. If the `spool` function signals an error during either flattening or unflattening, QuickDraw GX posts an `unflattening_interrupted_by_client` error. If the `spool` function attempts to call `GXFlattenShape`, QuickDraw GX posts a `procedure_not_reentrant` error.

ERRORS, WARNINGS, AND NOTICES

Errors

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>procedure_not_reentrant</code>	
<code>parameter_is_nil</code>	(debugging version)
<code>spoolProcedure_is_nil</code>	
<code>unflattening_interrupted_by_client</code>	
<code>parameter_out_of_range</code>	(debugging version)
<code>inconsistent_parameters</code>	(debugging version)

Warnings

<code>tags_of_type_flst_removed</code>	(debugging version)
--	---------------------

SEE ALSO

The `spool` block structure is described on page 2-49. The format for the application-defined `spool` function is described on page 2-91.

The format for the flattened data, including all opcodes, is described in the stream format chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

To convert a flattened shape back to its object-based format, use the `GXUnflattenShape` function, described in the next section.

GXUnflattenShape

You can use the `GXUnflattenShape` function to restore the object form of a shape—including all the objects that it references—from a stream-based description created by the `GXFlattenShape` function.

```
gxShape GXUnflattenShape(struct gxSpoolBlock *block, long count,
                        const gxViewPort portList[])
```

<code>block</code>	A pointer to a spool block structure. QuickDraw GX uses information in the spool block to unflatten the data.
<code>count</code>	The number of view ports in the view port list (the number of elements in the <code>portList</code> array).
<code>portList</code>	An array of references to view port objects. It is the list of view ports to assign to the transform object for the unflattened shape.

function result A reference to the newly created (unflattened) shape.

DESCRIPTION

The `GXUnflattenShape` function reconstructs a shape object and all its associated objects from stream data in a buffer pointed to by the spool block specified in the `block` parameter.

Before calling `GXUnflattenShape`, you need to allocate a spool block structure and buffer to hold the flattened data, and place a pointer to the buffer and a specification of its size into the spool block. You also place into the spool block a pointer to an application-defined spool function that reads the flattened data into the buffer. The spool function responds to commands from QuickDraw GX to open, read, and close the file containing the flattened data.

Note that unflattening a shape also causes creation of all its referenced objects, such as its style, ink, and transform, and all of their referenced objects. Unflattening a picture can cause the creation of many shape objects.

The flattened stream as read by `GXUnflattenShape` consists of a series of opcodes and associated data, following the QuickDraw GX stream format.

SPECIAL CONSIDERATIONS

If no error occurs, the `GXUnflattenShape` function creates one or more QuickDraw GX objects. You are responsible for disposing of those objects when you no longer need them.

If the spool function signals an error during either flattening or unflattening, QuickDraw GX posts an `unflattening_interrupted_by_client` error. If the spool function attempts to call `GXUnflattenShape`, QuickDraw GX posts a `procedure_not_reentrant` error.

ERRORS, WARNINGS, AND NOTICES**Errors**

out_of_memory	
procedure_not_reentrant	
parameter_is_nil	(debugging version)
spoolProcedure_is_nil	
unflattening_interrupted_by_client	
font_not_found	
parameter_out_of_range	(debugging version)
inconsistent_parameters	(debugging version)

Warnings

unrecognized_stream_version
bad_data_in_stream

SEE ALSO

The spool block structure is described on page 2-49. The format for the application-defined spool function is described on page 2-91.

The format for the flattened data, including all opcodes, is described in the stream format chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

To convert a QuickDraw GX shape to its flattened form, use the `GXFlattenShape` function, described in the previous section.

Application-Defined Spool Function

This section describes the interface to an application-defined function that can be used for saving and restoring shape objects.

MySpoolProc

The QuickDraw GX functions `GXFlattenShape` and `GXUnflattenShape` require the calling application to supply a pointer to a function that, respectively, saves flattened data or supplies data to be flattened. The flattening/unflattening spool function has the following interface:

```
long MySpoolProc(gxSpoolCommand command,
                struct gxSpoolBlock *block);
```

command	A selector with which QuickDraw GX specifies the operation the spool function is to perform.
block	A pointer to the spool block used for the current flattening or unflattening operation.
function result	Zero if the unflattening or flattening operation can continue; nonzero if QuickDraw GX must abort the operation.

DESCRIPTION

The purpose of the flattening/unflattening spool function is to move flattened data into or out of memory as instructed by QuickDraw GX. You place a pointer to the spool function in the appropriate field of the spool block structure that you allocate before calling `GXFlattenShape` or `GXUnflattenShape`.

Your spool function should respond to the `command` parameter and perform the appropriate operation. Constants for the recognized spool commands are defined in the `gxSpoolCommands` enumeration:

Constant	Value	Explanation
<code>gxOpenReadSpool</code>	1	The spool function is to open the flattened file for reading into the buffer used by <code>GXUnflattenShape</code> .
<code>gxOpenWriteSpool</code>	2	The spool function is to open a file for receiving data from the buffer used by <code>GXFlattenShape</code> .
<code>gxReadSpool</code>	3	The spool function is to read the data into the buffer for use by <code>GXUnflattenShape</code> .
<code>gxWriteSpool</code>	4	The spool function is to write the data placed into the buffer by <code>GXFlattenShape</code> to the file.
<code>gxCloseSpool</code>	6	The spool function is to close the file.

Your spool function's function result is a status indicator to QuickDraw GX. If the operation must be aborted (for example, if a file error occurs), return a nonzero value as the function result. Otherwise, return 0 and the flattening or unflattening operation can continue.

For simple flattening and unflattening, your spool function need only read and write the amounts of data specified by QuickDraw GX. However, you can also write a spool function that actually parses the data stream during flattening and unflattening; see Listing 2-5 on page 2-41 for an example.

SPECIAL CONSIDERATIONS

During a flattening or unflattening procedure, a memory error can cause a restart of the function, which might not necessarily reset the stream back to its original position. Therefore, your open, read, and write routines must always be sure to set the stream to the correct position in the buffer each time.

SEE ALSO

The spool block structure is described on page 2-49.

The `GXFlattenShape` function is described on page 2-88. The `GXUnflattenShape` function is described on page 2-90.

Summary of Shape Objects

Constants and Data Types

The Shape Object

```
typedef struct gxPrivateShapeRecord *gxShape;
```

Shape Type

```
enum gxShapeTypes {
    gxEmptyType = 1,      /* empty shape; contained by all geometries */
    gxPointType,         /* point shape */
    gxLineType,          /* line shape */
    gxCurveType,         /* curve shape */
    gxRectangleType,     /* rectangle shape */
    gxPolygonType,       /* polygon shape; can represent multiple polygons */
    gxPathType,          /* path shape; can include lines and curves */
    gxBitmapType,        /* bitmap shape */
    gxTextType,          /* text shape; single size, encoding, and style */
    gxGlyphType,         /* glyph shape; can use multiple text styles */
    gxLayoutType,        /* layout shape; can include linguistic info */
    gxFullType,          /* full shape; includes all geometries */
    gxPictureType        /* picture shape; can contain other shapes */
};
```

```
typedef long gxShapeType;
```

Shape Fill

```
enum gxShapeFills {
    gxNoFill,
    gxOpenFrameFill,
    gxFrameFill = gxOpenFrameFill,
    gxClosedFrameFill,
    gxHollowFill = gxClosedFrameFill,
    gxEvenOddFill,
    gxSolidFill = gxEvenOddFill,
    gxWindingFill,
    gxInverseEvenOddFill,
};
```

Shape Objects

```

    gxInverseSolidFill = gxInverseEvenOddFill,
    gxInverseFill = gxInverseEvenOddFill,
    gxInverseWindingFill
};

```

```
typedef long gxShapeFill;
```

Shape Attributes

```

enum gxShapeAttributes {
    gxNoAttributes,
    gxDirectShape          = 0x0001,    /* prefer shape data to be in GX heap */
    gxRemoteShape         = 0x0002,    /* prefer shape data on accelerator */
    gxCachedShape         = 0x0004,    /* pre-calculate to optimize drawing */
    gxLockedShape         = 0x0008,    /* prevent changes to shape's geometry */
    gxGroupShape          = 0x0010,    /* treat as one shape for hit-testing */
    gxMapTransformShape   = 0x0020,    /* alter transform, not shape geometry */
    gxUniqueItemsShape    = 0x0040,    /* copy items added to picture shapes */
    gxIgnorePlatformShape = 0x0080,    /* assume glyph, not character, codes */
    gxNoMetricsGridShape  = 0x0100,    /* draw without font scaler's hinting */
    gxDiskShape           = 0x0200,    /* unload this shape first */
    gxMemoryShape         = 0x0400     /* unload this shape last */
};

```

```
typedef long gxShapeAttribute;
```

Flatten Flags

```

enum gxFlattenFlags{
    gxFontListFlatten      = 0x01,     /* add a tag listing fonts used */
    gxFontGlyphsFlatten   = 0x02,     /* add a tag listing glyphs used */
    gxFontVariationsFlatten = 0x04,    /* add a tag for font variations */
    gxBitmapAliasFlatten  = 0x08      /* flatten all bitmap image data */
};

```

```
typedef long gxFlattenFlag;
```

The Spool Block Structure

```

struct gxSpoolBlock {
    /* these fields are read from (but not written to) by QuickDraw GX */
    gxSpoolProcPtr    spoolProcedure; /* pointer to spool function */
    void              *buffer;        /* pointer to application buffer */
    long              bufferSize;     /* bytes for QuickDraw GX to use */
};

```


Shape Objects

```

/* these fields are written to (but not read from) by QuickDraw GX */
long          count;          /* bytes for app to read/write */
long          operationSize;  /* size including operand byte */
long          operationOffset; /* offset within current operation */
gxGraphicsOpcode lastTypeOpcode; /* type of last created object */
gxGraphicsOpcode currentOperation; /* last op. emitted or interpreted */
gxGraphicsOpcode currentOperand; /* such as gxTransformTypeOpcode */
unsigned char compressed;     /* a gxTwoBitCompressionValues */
};

```

The Hit-Test Info Structure

```

struct gxHitTestInfo {
    gxShapePart    what;
    long           index;
    Fixed          distance;
    gxShape        which;
    gxShape        containerPicture;
    long           containerIndex;
    long           totalIndex;
};

```

Spool Commands

```

enum gxSpoolCommands {
    gxOpenReadSpool= 1,
    gxOpenWriteSpool,
    gxReadSpool,
    gxWriteSpool,
    gxCloseSpool,
};

typedef long gxSpoolCommand;

```

Functions

Creating and Manipulating Shape Objects

```

gxShape GXGetDefaultShape    (gxShapeType aType);
void GXSetDefaultShape      (gxShape target);
gxShape GXNewShape          (gxShapeType aType);
void GXDisposeShape         (gxShape target);
long GXGetShapeSize         (gxShape source);

```

Shape Objects

```

gxShape GXCopyToShape      (gxShape target, gxShape source);
gxShape GXCopyDeepToShape  (gxShape target, gxShape source);
boolean GXEqualShape       (gxShape one, gxShape two);
gxShape GXCloneShape       (gxShape source);
void GXCacheShape          (gxShape source);
void GXDisposeShapeCache   (gxShape target);
long GXGetShapeCacheSize   (gxShape source);

```

Manipulating Shape Object Properties

```

gxShapeType GXGetShapeType (gxShape source);
void GXSetShapeType        (gxShape target, gxShapeType newType);
void GXSetShapeGeometry    (gxShape target, gxShape geometry);
gxShapeFill GXGetShapeFill (gxShape source);
void GXSetShapeFill        (gxShape target, gxShapeFill newFill);
gxStyle GXGetShapeStyle    (gxShape source);
void GXSetShapeStyle        (gxShape target, gxStyle newStyle);
gxInk GXGetShapeInk        (gxShape source);
void GXSetShapeInk         (gxShape target, gxInk newInk);
gxTransform GXGetShapeTransform
                           (gxShape source);
void GXSetShapeTransform   (gxShape target, gxTransform newTransform);
gxShapeAttribute GXGetShapeAttributes
                           (gxShape source);
void GXSetShapeAttributes  (gxShape target, gxShapeAttribute attributes);
void GXResetShape          (gxShape target);
long GXGetShapeOwners      (gxShape source);
long GXGetShapeTags        (gxShape source, long tagType, long index,
                           long count, gxTag items[]);
void GXSetShapeTags        (gxShape target, long tagType, long index,
                           long oldCount, long newCount,
                           const gxTag items[]);

```

Directly Manipulating Shape Geometry

```
void GXLockShape          (gxShape target);  
void GXUnlockShape       (gxShape target);  
void *GXGetShapeStructure (gxShape source, long *length);  
void GXChangedShape     (gxShape target);
```

Drawing and Hit-Testing Shapes

```
void GXDrawShape          (gxShape source);  
gxShapePart GXHitTestShape (gxShape target, const gxPoint *test,  
                             gxHitTestInfo *result);
```

Flattening and Unflattening Shapes

```
void GXFlattenShape       (gxShape source, gxFlattenFlag flags,  
                             gxSpoolBlock *block);  
gxShape GXUnflattenShape  (struct gxSpoolBlock *block, long count,  
                             const gxViewPort portList[]);
```

Application-Defined Spool Function

```
long MySpoolProc          (gxSpoolCommand command,  
                             struct gxSpoolBlock *block);
```


Style Objects

Contents

About Style Objects	3-3
Style Object Properties	3-4
The Default Style Object	3-6
Using Style Objects	3-7
Creating and Manipulating Style Objects	3-7
Creating and Deleting a Style Object	3-7
Copying, Comparing, and Cloning Style Objects	3-8
Loading and Unloading Style Objects	3-10
Manipulating Style Object Properties	3-10
Resetting a Style Object's Default Properties	3-11
Getting and Setting Style Attributes and Text Attributes	3-11
Manipulating a Style Object's Owner Count	3-11
Getting and Setting a Style Object's Tag References	3-14
Style-Related Functions Described Elsewhere	3-14
Style Objects Reference	3-15
Constants and Data Types	3-16
The Style Object	3-16
Functions	3-16
Creating and Manipulating Style Objects	3-16
GXNewStyle	3-17
GXDisposeStyle	3-17
GXCopyToStyle	3-18
GXEqualStyle	3-19
GXCloneStyle	3-20
Manipulating Style Object Properties	3-21
GXResetStyle	3-21
GXGetStyleOwners	3-22
GXGetStyleTags	3-22
GXSetStyleTags	3-24

CHAPTER 3

Summary of Style Objects	3-26
Constants and Data Types	3-26
Functions	3-26

Style Objects

This chapter describes style objects and the functions you can use to manipulate them. Read this chapter if you create or use any kind of style objects for the QuickDraw GX shapes you create.

Before reading this chapter, you should be familiar with the information in the chapter “Introduction to QuickDraw GX” in this book. You should also be familiar with shape objects, as discussed in the chapter “Shape Objects” in this book.

For more information on style objects for graphic shapes, see the geometric styles chapter of *Inside Macintosh: QuickDraw GX Graphics*. For more information on style objects for typographic shapes, see the typographic styles chapter of *Inside Macintosh: QuickDraw GX Typography*.

This chapter introduces QuickDraw GX style objects and describes their properties. It then shows how to use general QuickDraw GX style-manipulation functions to

- n create and manipulate style objects
- n manipulate style object properties

This chapter also lists and cross-references all style-related QuickDraw GX functions that are described elsewhere in this book and in other parts of *Inside Macintosh*.

About Style Objects

A style object exists to provide information about a shape. Each QuickDraw GX shape consists of a shape object, a style object, an ink object, and a transform object; the style object associated with a shape defines much of the shape’s appearance, such as the size of the pen with which it is drawn or the size of its text.

QuickDraw GX identifies an individual style object through a style **reference**. To obtain information about a style object, you must send its reference as a parameter to a QuickDraw GX function (except that you can determine if two references identify the same style object simply by comparing them for equality, and you can examine a reference to see if it is `nil`).

Styles are device independent. Their information is not affected by the properties of the display device to which the shapes they modify are drawn.

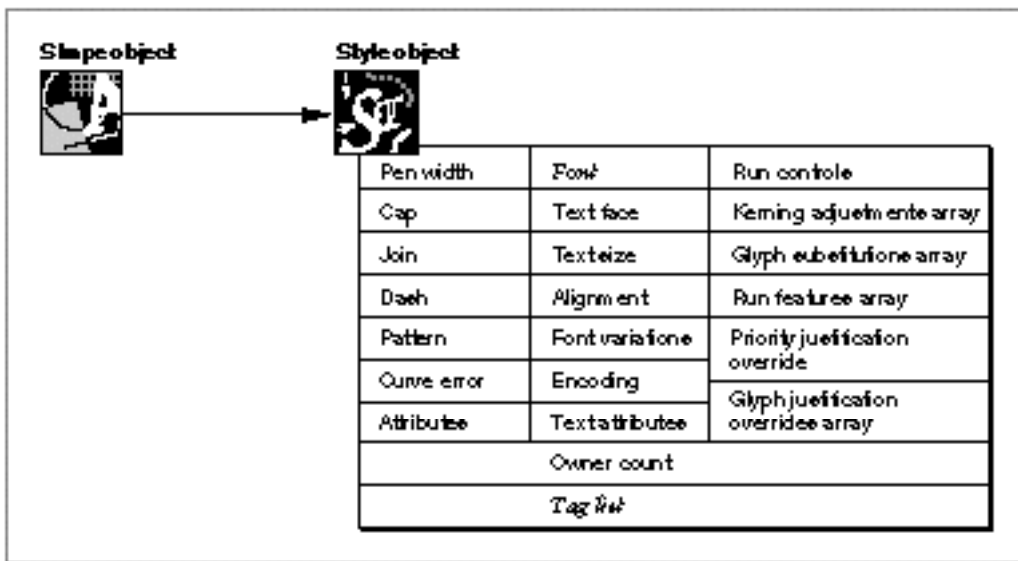
There are three categories of information that style objects contain: graphic, typographic, and common. The graphic information applies to style objects associated with graphic shapes, the typographic information applies to style objects associated with typographic shapes, and the common information applies to both. Because the information is stored separately, the same style object can apply to both kinds of shapes. The QuickDraw GX object architecture allows you to perform several operations on a style object without regard for what kind it is; those are the operations described in this chapter. Features and operations specific to styles for graphic shapes are described in *Inside Macintosh: QuickDraw GX Graphics*; those specific to styles for typographic shapes are described in *Inside Macintosh: QuickDraw GX Typography*.

Style Object Properties

The interface to style objects is entirely procedural. You manipulate the information in a style object by modifying its properties using QuickDraw GX functions.

Style objects have 22 accessible properties, as shown in Figure 3-1. The properties are grouped into columns that reflect the category of shape that uses them. Note that, because a style is an object and not a data structure, the order of the properties as shown in Figure 3-1 is completely arbitrary. Properties in italics are references to other objects.

Figure 3-1 The style object and its properties



Seven properties pertain mostly to style objects associated with graphic shapes:

- n **Pen width.** The width of the pen used to draw the shape.
- n **Cap.** The shape (such as an arrowhead, or any other geometric shape) to draw at the start and end of each contour in the shape.
- n **Join.** The appearance (such as rounded or sharp, or any other geometric shape) of corners where a shape's lines or contours meet.
- n **Dash.** The appearance of dashed lines or contours in a shape. The dashing capability is very general in QuickDraw GX; you can specify any geometric shape, or even a sequence of glyphs, for a dash.
- n **Pattern.** The pattern (actually, any geometric shape, glyph shape, or bitmap shape) to use in filling the geometry of the shape.
- n **Curve error.** The allowable error for operations such as converting a path shape to a polygon shape.

Style Objects

- n **Attributes.** A set of flags that allow you to specify how QuickDraw GX places the pen and whether the shape is constrained to a grid when drawn. (The grid-constraining attributes can apply to typographic shapes also.)

Thirteen of the style object's properties pertain only to styles associated with typographic shapes. The portion of a typographic shape to which a style object applies is called a **style run**. The first seven typographic style properties apply, for the most part, to all typographic shapes:

- n **Font.** The reference to the font to use in drawing the text of this style run. (In QuickDraw GX, a font is an object.)
- n **Text face.** The text face—the constructed stylistic variation from plain text—to apply when drawing the text of this style run.
- n **Text size.** The size, in typographic points (72 per inch), to draw the text of this style run.
- n **Alignment.** The alignment value to use when drawing the text of this style run. Text may be left-aligned, right-aligned, anywhere between the two alignments (such as centered), or fully justified. (This property is not used by layout shapes).
- n **Font variations.** The list of font variations—stylistic variations built into the font—specified for drawing the text of this style run.
- n **Encoding.** The type of character encoding used to represent the text of this style run, as well as its script and language.
- n **Text attributes.** A set of flags that allow you to specify how QuickDraw GX alters glyph outlines or chooses the proper metrics for horizontal or vertical text.

The remaining six of the thirteen typographic style properties apply to layout shapes only:

- n **Run controls.** A set of values and flags that control various aspects of how the text in this style run is displayed.
- n **Kerning adjustments array.** An array specifying changes to the font-specified kerning (positional adjustment) for pairs of glyphs in this style run.
- n **Glyph substitutions array.** An array specifying substitute glyphs for those that would normally be displayed in this style run.
- n **Run features array.** An array specifying the set of font features—typographic capabilities as defined by the font—to apply to the text of this style run.
- n **Priority justification override.** A structure that redefines the justification priorities and behaviors for whole classes of glyphs.
- n **Glyph justification overrides array.** An array that redefines the justification priorities and behaviors for individual glyphs.

Style Objects

The two remaining style object properties pertain to all styles, for all shapes:

- n **Owner count.** The number of existing references to this style object.
- n **Tag list.** A list of references to custom information about this style object, stored in private data structures called *tag objects*. The chapter “Tag Objects” in this book describes tag objects in general and how you can use them to add custom information to objects.

QuickDraw GX provides functions to manipulate each of these style object properties. Table 3-1 shows where to go for that information, depending on the type of shape object that uses the style.

Table 3-1 Where to go for information on style object properties and functions

For style objects used by...	Look in...
Graphic shapes	Geometric styles chapter of <i>QuickDraw GX Graphics</i>
All typographic shapes	Typographic styles chapter of <i>QuickDraw GX Typography</i> (For style attributes that can apply to typographic shapes, see also the geometric styles chapter of <i>QuickDraw GX Graphics</i>)
Layout shapes only	Layout styles and layout line control chapters of <i>QuickDraw GX Typography</i>
All shapes	This chapter

As Table 3-1 shows, most style-object properties and functions are described elsewhere. Only those properties that pertain to all shapes—the owner count and tag list, and the functions that manipulate them—are described in this chapter.

The Default Style Object

When QuickDraw GX first creates a style object, that object has default characteristics defined by QuickDraw GX. Every default style object has the following properties:

- n an empty tag list
- n an owner count of 1

All other properties are zero or *nil*, except that the value of the text size property is 12.0, and the scale value within the dash property is 1.0. The font property is *nil*, which means that QuickDraw GX uses the default font in drawing text; however, your application can control what font is used for the default. See the font objects chapter in *Inside Macintosh: QuickDraw GX Typography* for more information.

Unlike shape objects, whose default properties vary with type, there is only one single default style object for QuickDraw GX. If the shape objects you create reference the default style object, you need to explicitly set all graphic or typographic properties for that style after you create the shape. Also unlike shape objects, you cannot change the definition of the default style object. However, you can create a style object with specific properties, and then change the definition of the default shape object so that newly created shapes reference that customized style object.

Using Style Objects

This section describes the basic style-creation and style-manipulation capabilities that QuickDraw GX provides, capabilities that are independent of the specific type of style object involved. For detailed information on using styles of specific types, see the appropriate chapters of *Inside Macintosh: QuickDraw GX Graphics* and *Inside Macintosh: QuickDraw GX Typography*.

This section describes how you can

- n create and manipulate style objects
- n manipulate certain style object properties

Creating and Manipulating Style Objects

This section describes how you can create and interact with style objects as whole entities—to create, dispose of, copy, compare, and clone them. Manipulating the individual properties of style objects is described under “Manipulating Style Object Properties” beginning on page 3-10.

Creating and Deleting a Style Object

QuickDraw GX provides the `GXNewStyle` function to allow you to create a new style object. Before you can create a style object, you need to be in the QuickDraw GX environment. However, if you are not already in the QuickDraw GX environment, `GXNewStyle` calls the necessary functions for you. The functions for controlling memory use in the QuickDraw GX environment are described in the memory management chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

Note that you can also create a new style object by copying an existing one: see the section “Copying, Comparing, and Cloning Style Objects” beginning on page 3-8.

To delete your application’s reference to a style object, call the `GXDisposeStyle` function. Calling `GXDisposeStyle` may or may not actually release the memory allocated for that style object, depending on the style’s owner count. `GXDisposeStyle` decreases the style object’s owner count by 1; if that brings the owner count to zero, the style is completely deleted and its memory released. See “Manipulating a Style Object’s Owner Count” beginning on page 3-11.

Style Objects

Owner counts and what it means to dispose of an object are described in general in the chapter “Introduction to Objects” in this book.

The following code fragment defines and creates the style object `myStyle`, sets some of its properties, and disposes of it:

```
gxStyle      myStyle;
.
.
.
myStyle = GXNewStyle ();
GXSetStylePen(myStyle, ff(2));
GXSetStyleAttributes(myStyle, gxOutsideFrameStyle);
.
.
.
if (myStyle != nil) GXDisposeStyle(myStyle);
```

The `GXNewStyle` function is described on page 3-17. The `GXDisposeStyle` function is described on page 3-17.

Copying, Comparing, and Cloning Style Objects

You can use the `GXCopyToStyle` function to copy information from one style object to another or to create a new copy of a style object.

Listing 3-1 is a code fragment that changes the type of the shape `myShape` to a glyph shape with four distinct style runs, and then fills out the style list, an array of style-object references in the geometry of the glyph shape. (Glyph shapes and layout shapes have style-object references in their geometries in addition to the style property that every shape object has.) The code creates new copies of the style object originally referenced by `myShape`, each time assigning the style reference to a position in the `styleSet` array and then modifying some of the style’s properties. Finally, the code assigns the style list to the shape geometry.

The code in Listing 3-1 uses the library functions `SetStyleCommonFont` and `SetStyleCommonFace` to modify the font and text-face properties of the style objects it creates by copying. The text is defined in the string `str`, and the lengths of the style runs are defined in the `runs` array. (Each style run is defined to be one glyph long in this sample.)

Listing 3-1 Building a style list by copying a style object

```

GXSetShapeType(myShape, gxGlyphType);

/* use the default shape's style for first style run */
styleSet[0] = nil;

/* use condensed Helvetica for the second style run */
styleSet[1] = GXCopyToStyle(nil, GXGetShapeStyle(myShape));
SetStyleCommonFont(styleSet[1], helveticaFont);
SetStyleCommonFace(styleSet[1], gxCondense);

/* use extended Times for the third style run */
styleSet[2] = GXCopyToStyle(nil, GXGetShapeStyle(myShape));
SetStyleCommonFont(styleSet[2], timesFont);
SetStyleCommonFace(styleSet[2], gxExtend);

/* use 20-pt. italic Helvetica for the fourth style run */
styleSet[3] = GXCopyToStyle(nil, GXGetShapeStyle(myShape));
SetStyleCommonFont(styleSet[3], helveticaFont);
SetStyleCommonFace(styleSet[3], gxItalic);
GXSetStyleTextSize(styleSet[3], ff(20));

/* set the size (number of glyphs) of each style run */
for (counter = 0; counter < strlen(str); counter++) {
    runs[counter] = 1;          /* each run is 1 glyph long */
    styles[counter] = styleSet[counter & 3];
}
/* assign the styles array to the style list */
GXSetGlyphs(myShape, nil, nil, nil, nil, runs, styles);

```

You can test if two style-object references refer to the same style object by simply testing the references for equality. You can also compare two different style objects for equality with the `GXEqualStyle` function. For two style objects to be equal, their graphic and typographic properties must have identical values, although their general object properties (owner count and tag list) do not need to be identical. Note that style object copies created with the `GXCopyToStyle` function are always equal to the style from which they were copied.

In certain circumstances, you may want to copy a reference to a style object without actually copying the style object. For example, you may want two variables to refer to the same style object, so that editing one of them affects both. This is called **cloning** a style, rather than copying a style. You can use the `GXCloneStyle` function to clone a style object.

Style Objects

Functionally, `GXCloneStyle` does nothing more than increase the owner count of a style object. You can clone a style with a statement such as the following:

```
aStyleClone = GXCloneStyle(aStyle);
```

This code has almost the same effect as

```
aStyleClone = aStyle;
```

that is, it sets the `aStyleClone` variable to reference the same style object as the `aStyle` variable. The difference is that `GXCloneStyle` also increments the style's owner count.

For more information about cloning objects, see the chapter “Introduction to QuickDraw GX” in this book. For information on manipulating style owner counts, including examples of cloning styles, see the section “Manipulating a Style Object's Owner Count” beginning on page 3-11 of this chapter.

The `GXCopyToStyle` function is described on page 3-18. The `GXEqualStyle` function is described on page 3-19. The `GXCloneStyle` function is described on page 3-20.

Loading and Unloading Style Objects

Although you rarely need to, you can influence memory-allocation decisions involving objects that you have created. If your application needs to have a style object in memory, you can force QuickDraw GX to load it into memory. When your application no longer needs the style object in a loaded state, you can instruct QuickDraw GX to unload it.

You call the `GXLoadStyle` function to make sure that a style object is in memory; if necessary, QuickDraw GX brings the object into memory from an unloaded state. You can call the `GXUnloadStyle` function to instruct QuickDraw GX that it is free to unload the style object at any time. These functions are described in the memory management chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

Manipulating Style Object Properties

Once you have created a style object, you can customize some of its features using the techniques described in this section. However, most of the functions you use to set style properties are described in the chapters that discuss style objects in *Inside Macintosh: QuickDraw GX Graphics* and *Inside Macintosh: QuickDraw GX Typography*.

This section describes how to manipulate those properties of style objects that are independent of the type of shape the style is associated with. You can reset a style's properties back to their default values, you can determine the owner count, and you can get and set the tag list.

For manipulating style objects as a whole, see “Creating and Manipulating Style Objects” beginning on page 3-7.

Resetting a Style Object's Default Properties

When you create a new style object with the `GXNewStyle` function, QuickDraw GX creates a style object with default properties. If you have altered any of the style object's properties using functions described in this chapter or in *Inside Macintosh: QuickDraw GX Graphics* or *Inside Macintosh: QuickDraw GX Typography*, you can reset the properties back to their default values using the `GXResetStyle` function.

Calling `GXResetStyle` returns all of the style's graphic and typographic properties to their default values. It does not affect the style's owner count or tag list.

The `GXResetStyle` function is described on page 3-21.

Getting and Setting Style Attributes and Text Attributes

A style object has two separate properties, *attributes* and *text attributes*, that consist of flags that affect style behavior. The *attributes* property of a style object affects mostly graphic shapes. You retrieve and assign attribute values, such as pen width, with the `GXGetStyleAttributes` and `GXSetStyleAttributes` functions. These functions, and the style attributes themselves, are described in the geometric styles chapter of *Inside Macintosh: QuickDraw GX Graphics*.

The *text attributes* property of a style object affects typographic shapes only. You retrieve and assign text attribute values, such as vertical-text selection, with the `GXGetStyleTextAttributes` and `GXSetStyleTextAttributes` functions, respectively. These functions, and the text attributes themselves, are described in the typographic styles chapter of *Inside Macintosh: QuickDraw GX Typography*.

Manipulating a Style Object's Owner Count

The owner count of an object indicates the number of current references to that object. In general, QuickDraw GX manages owner counts for you. For example, when you create a new style object, QuickDraw GX sets the owner count of the new style to 1. When you assign an existing style object to a shape, QuickDraw GX increments the style's owner count to correspond to the new reference to the style contained in the shape object.

If you want to manage a style's owner count directly—for example, if you want to track object references that you place in your own data structures, or if you want to know whether a style object is shared—you can use the `GXGetStyleOwners` function to determine the owner count of a style, and the `GXCloneStyle` and `GXDisposeStyle` functions to change the owner count of a style. The `GXCloneStyle` function increments the style's owner count, and the `GXDisposeStyle` function decrements the style's owner count, freeing the memory used by the style if the owner count goes to 0.

The `GXGetStyleOwners` function is described on page 3-22.

The following subsections discuss two common owner-count problems and how to avoid them. The problems are discussed in terms of style objects, but they apply equally well to other shared objects.

Avoiding Excessive Owner Count

The following is one plausible, but incorrect, way to create a style object and assign it to (the style reference property of) a shape:

```
GXSetShapeStyle(myShape, GXNewStyle());
```

After the execution of this statement, the owner count of the just-created style object is 2, not 1; creating the style object initialized its owner count to 1, and assigning it to the shape incremented its owner count to 2. If you were unaware of that, and deleted the shape object with the statement

```
GXDisposeShape(myShape);
```

the owner count of the style object would be decremented to 1, and the style would be left allocated in the heap when it should have been deleted.

A better way to create and assign a style object is to allocate a variable and use it in the assignment:

```
myStyle = GXNewStyle();
GXSetShapeStyle(myShape, myStyle);
```

As before, the style object's owner count is now 2. When you are finished with the variable reference to the style object, you can dispose of it:

```
GXDisposeStyle(myStyle);
```

That decreases the style's owner count to 1. When you are finished with the shape object, dispose of it as before:

```
GXDisposeShape(myShape);
```

That decreases the style's owner count to 0, and the style object is deleted as intended.

Avoiding Insufficient Owner Count

The following is one plausible, but incorrect, way to temporarily assign a style object to a shape, referenced in this example by the variable `myShape`. These statements save the original style into a variable, create a new style object, and assign the new style to the shape:

```
gxStyle myOldStyle = GXGetShapeStyle(myShape);
gxStyle myNewStyle = GXNewStyle();
GXSetShapeStyle(myShape, myNewStyle);
```


Style Objects

The first statement does not increase the owner count of the style referenced by `myOldStyle`; no new object is created and no additional references to `myShape` exist in any object. The second statement results in an owner count of 1 for the style referenced by `myNewStyle`. The third statement decrements the owner count of the style referenced by `myOldStyle`, and increments the owner count of the style referenced by `myNewStyle` (from 1 to 2).

Now suppose that you manipulate the new style object, draw the shape, and then wish to dispose of the new style and reassign the original style object back to the shape. You might expect to make two statements like this:

```
GXDisposeStyle(myNewStyle);
GXSetShapeStyle(myShape, myOldStyle);
```

As you would expect, disposing of `myNewStyle` decrements the owner count of the new style object from 2 to 1, and calling `GXSetShapeStyle` further decrements the owner count of the new style from 1 to 0, so that QuickDraw GX can delete it. However, the original style object, referenced by `myOldStyle`, may have been deleted by the original call to `GXSetShapeStyle` (because its owner count may have gone to 0 as a result of the call). If it has, `myOldStyle` will be `nil` and the new call to `GXSetShapeStyle` will fail.

A better way to temporarily save and restore a style object is to clone the original style before assigning the new style, as follows:

```
gxStyle myOldStyle = GXGetShapeStyle(myShape);
gxStyle myNewStyle = GXNewStyle();
GXCloneStyle(myOldStyle);
GXSetShapeStyle(myShape, myNewStyle);
```

The result of these statements is (assuming no other references to the style objects) an owner count of 2 for both the original and new style objects. Then, when the time comes to restore the original style object to the shape, you can make these statements:

```
GXDisposeStyle(myNewStyle);
GXSetShapeStyle(myShape, myOldStyle);
GXDisposeStyle(myOldStyle);
```

The first statement decrements the owner count of the new style from 2 to 1; the second statement decrements it from 1 to 0. The second statement increments the owner count of the original style from 1 to 2, so the third statement is added to bring it back down to 1, its original value.

Getting and Setting a Style Object's Tag References

You can examine the list of references to tag objects currently associated with a style object using the `GXGetStyleTags` function. Once you create a tag object, you can attach it to a style object using the `GXSetStyleTags` function. You can attach as many tag objects as you like to a style object.

Tag objects and the basic functions for manipulating them are described in the chapter “Tag Objects” in this book. That chapter also lists the common tag types defined and reserved by Apple Computer, Inc.

The `GXGetStyleTags` function is described on page 3-22. The `GXSetStyleTags` function is described on page 3-24.

Style-Related Functions Described Elsewhere

Table 3-2 lists functions whose names contain the word `Style` that are either not described in this chapter or are described in more detail elsewhere. For each book and chapter, the table lists the style-related functions described in that chapter.

Table 3-2 Style-related functions described elsewhere

Book and chapter	Functions described
<i>Inside Macintosh: QuickDraw GX Graphics</i> “Geometric Styles”	<code>GXGetStylePen</code> <code>GXSetStylePen</code> <code>GXGetStyleCap</code> <code>GXSetStyleCap</code> <code>GXGetStyleJoin</code> <code>GXSetStyleJoin</code> <code>GXGetStyleDash</code> <code>GXGetStyleDash</code> <code>GXGetStylePattern</code> <code>GXSetStylePattern</code> <code>GXGetStyleCurveError</code> <code>GXSetStyleCurveError</code> <code>GXGetStyleAttributes</code> <code>GXSetStyleAttributes</code>

Table 3-2 Style-related functions described elsewhere (continued)

Book and chapter	Functions described
<i>Inside Macintosh: QuickDraw GX Typography</i>	
“Typographic Styles”	GXGetStyleFont GXSetStyleFont GXGetStyleFontMetrics GXGetStyleFace GXSetStyleFace GXGetStyleTextSize GXSetStyleTextSize GXGetStyleJustification GXSetStyleJustification GXGetStyleFontVariations GXSetStyleFontVariations GXGetStyleFontVariationSuite GXGetStyleEncoding GXSetStyleEncoding GXGetStyleTextAttributes GXSetStyleTextAttributes
“Layout Styles”	GXGetStyleRunControls GXSetStyleRunControls GXGetStyleRunKerningAdjustments GXSetStyleRunKerningAdjustments GXGetStyleRunGlyphSubstitutions GXSetStyleRunGlyphSubstitutions GXGetStyleRunFeatures GXSetStyleRunFeatures
“Layout Line Control”	GXGetStyleRunPriorityJustOverride GXSetStyleRunPriorityJustOverride GXGetStyleRunGlyphJustOverrides GXSetStyleRunGlyphJustOverrides

Style Objects Reference

This section provides reference information about the data structures and functions that allow you to create and manipulate style objects and alter their properties. It includes

- n a type definition of the data type that applies to style objects in general
- n descriptions of the QuickDraw GX functions that operate on style objects in general, independent of the type of shape involved

Constants and Data Types

This section describes the data type that you use to gain access to style objects.

Style-related QuickDraw GX constants and data types not described in this section are related to geometric and typographic shapes, and are thus described in the geometric styles chapter of *Inside Macintosh: QuickDraw GX Graphics* and the typographic styles, layout styles, and layout line control chapters of *Inside Macintosh: QuickDraw GX Typography*.

The Style Object

QuickDraw GX provides you with access to an individual style object through a `gxStyle` reference:

```
typedef struct gxPrivateStyleRecord *gxStyle;
```

In this type definition, `gxStyle` is a type-checked reference, not an actual pointer to any defined structure. The contents of the style object are private.

Functions

This section describes the QuickDraw GX functions you can use to

- n create and manipulate a style object
- n manipulate the general object properties of a style object

Note

Style-related QuickDraw GX functions not described in this section are described in the chapters listed and cross-referenced in Table 3-2 on page 3-14. u

Creating and Manipulating Style Objects

This section describes the functions that manipulate styles as objects in memory. With the functions in this section, you can create, dispose of, copy, compare, and clone style objects.

To associate a style object with a QuickDraw GX shape object, use the `GXGetShapeStyle` and `GXSetShapeStyle` functions, described in the chapter “Shape Objects” in this book.

GXNewStyle

You can use the `GXNewStyle` function to create a new style object with default properties.

```
gxStyle GXNewStyle(void);
```

function result A reference to a newly created copy of the default style object.

DESCRIPTION

The `GXNewStyle` function creates a style object with an owner count of 1. All other properties of the style are set to their default values:

- n An empty tag list.
- n An owner count of 1.
- n A text size of 12.0.
- n A scale value within the dash property of 1.0.
- n No font specified.

All other properties are zero or `nil`.

SPECIAL CONSIDERATIONS

If no error occurs, the `GXNewStyle` function creates a style object; you are responsible for disposing of that object when you no longer need it.

ERRORS, WARNINGS, AND NOTICES

Errors
`out_of_memory`

SEE ALSO

Default style values are described in the section “The Default Style Object” on page 3-6.

GXDisposeStyle

You can use the `GXDisposeStyle` function to release a reference to a style object.

```
void GXDisposeStyle(gxStyle target);
```

`target` A reference to the style object to dispose of.

DESCRIPTION

The `GXDisposeStyle` function decrements the owner count of the style specified by the `target` parameter and releases any memory used by the style if the owner count goes to 0.

ERRORS, WARNINGS, AND NOTICES**Errors**

`style_is_nil`

SEE ALSO

Owner counts for style objects are discussed in the section “Copying, Comparing, and Cloning Style Objects” on page 3-8, and in the section “Manipulating a Style Object’s Owner Count” beginning on page 3-11. To examine the owner count of a style, use the `GXGetStyleOwners` function, described on page 3-22.

GXCopyToStyle

You can use the `GXCopyToStyle` function to create a copy of an existing style object.

```
gxStyle GXCopyToStyle(gxStyle target, gxStyle source);
```

`target` A reference to the style object to copy the source style object’s contents into. If you specify `nil` for this parameter, this function creates a new style object.

`source` A reference to the style object whose contents you want to copy.

function result A reference to the copy (that is, the target style).

DESCRIPTION

The `GXCopyToStyle` function copies the contents of an existing style object to another, or it creates a new style object and copies the contents of an existing style object into it. The function copies the stylistic and typographic properties and the tag list (but not the owner count) of the style object specified by the `source` parameter into the style object specified by the `target` parameter. It clones, but does not copy, the tag objects in the tag list.

If you specify `nil` for the `target` parameter, the `GXCopyToStyle` function creates a new style object and copies the source properties, including tag list, into it. The function gives the new style object an owner count of 1.

You can use the `GXCopyToStyle` function to create a copy of a style object in order to modify it without changing the original.

SPECIAL CONSIDERATIONS

If you specify `nil` for the `target` parameter and no error occurs, the `GXCopyToStyle` function creates a style object; you are responsible for disposing of that object when you no longer need it.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`style_is_nil`

SEE ALSO

To create a new style that is a copy of the default style instead of a copy of an existing style, use the `GXNewStyle` function, described on page 3-17.

To compare two style objects, use the `GXEqualStyle` function, described in the next section.

GXEqualStyle

You can use the `GXEqualStyle` function to determine if two style objects are equal.

```
boolean GXEqualStyle(gxStyle one, gxStyle two);
```

`one` A reference to one of the style objects to test for equality.

`two` A reference to the other style object to test for equality.

function result `true` if the style specified by the `one` parameter is equal to the style specified by the `two` parameter; `false` otherwise.

DESCRIPTION

The `GXEqualStyle` function returns as its function result a Boolean value indicating whether the two style objects are equal.

For two style objects to be equal, they must have identical properties, except that their common object properties (owner count and tag list) need not be identical.

ERRORS, WARNINGS, AND NOTICES**Errors**

out_of_memory
style_is_nil

SEE ALSO

To make a copy of a style object that is equal by the criteria of this function, use the `GXCopyToStyle` function, described in the previous section.

GXCloneStyle

You can use the `GXCloneStyle` function to clone a style object—that is, to add a reference to it and increment its owner count.

```
gxStyle GXCloneStyle(gxStyle source);
```

`source` A reference to the style to clone.

function result A reference to the cloned style.

DESCRIPTION

The `GXCloneStyle` function increments the owner count of the style referenced in the `source` parameter. You typically use this function when you want to create another reference to an existing style rather than creating a distinct copy of the style.

This function returns as its function result a reference to the style—the same reference you pass in as the `source` parameter. The only other action that `GXCloneStyle` performs is to increment the style's owner count.

ERRORS, WARNINGS, AND NOTICES**Errors**

style_is_nil

SEE ALSO

Owner counts for style objects are discussed in the section “Copying, Comparing, and Cloning Style Objects” beginning on page 3-8, and in the section “Manipulating a Style Object's Owner Count” beginning on page 3-11.

To examine the owner count of a style, use the `GXGetStyleOwners` function, described on page 3-22. To decrement the owner count of a style, use the `GXDisposeStyle` function, described on page 3-17.

Manipulating Style Object Properties

This section describes the functions that allow you to manipulate certain properties of style objects—those properties that are independent of the kind of style object. The functions in this section allow you to reset some of the properties of a style object to their default values, find a style object's owner count, and manipulate a style object's tag list.

Functions that allow you to manipulate graphics-specific style properties are described in the geometric styles chapter of *Inside Macintosh: QuickDraw GX Graphics*; functions that allow you to manipulate typographic-specific style properties are described in the typographic styles chapter and layout styles chapter of *Inside Macintosh: QuickDraw GX Typography*.

GXResetStyle

You can use the `GXResetStyle` function to reset the properties of an existing style object to their default values.

```
void GXResetStyle(gxStyle target);
```

`target` A reference to the style object whose properties you want to reset.

DESCRIPTION

The `GXResetStyle` function resets all properties of the target style object, except owner count and tag list, to their default values. The owner count and tag list are unaffected.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`style_is_nil`

SEE ALSO

Default properties of style objects are discussed in the section “The Default Style Object” on page 3-6.

GXGetStyleOwners

You can use the `GXGetStyleOwners` function to determine the number of references to a particular style.

```
long GXGetStyleOwners(gxStyle source);
```

`source` A reference to the style to find the owner count of.

function result The owner count of the source style.

DESCRIPTION

The `GXGetStyleOwners` function returns as its function result the owner count of the style specified by the `source` parameter. The owner count is the current number of references to the style object.

ERRORS, WARNINGS, AND NOTICES

Errors

`style_is_nil`

SEE ALSO

Owner counts are discussed in the section “Copying, Comparing, and Cloning Style Objects” on page 3-8, and in the section “Manipulating a Style Object’s Owner Count” beginning on page 3-11.

To increment the owner count of a style, use the `GXCloneStyle` function, described on page 3-20. To decrement the owner count of a style, use the `GXDisposeStyle` function, described on page 3-17.

GXGetStyleTags

You can use the `GXGetStyleTags` function to examine one or more of the tag objects associated with a style object.

```
long GXGetStyleTags(gxStyle source, long tagType, long index,
                    long count, gxTag items[]);
```

`source` A reference to the style object to examine the tag list of.

`tagType` The type of tag object to search for. A value of 0 indicates that you want to look for all tag types.

Style Objects

<code>index</code>	The (1-based) index of the first such tag reference to return.
<code>count</code>	The number of tag references to return.
<code>items</code>	An array to hold the returned tag references.

function result The number of tag references found that fit the criteria.

DESCRIPTION

The `GXGetStyleTags` function searches the tag list of the source style object for references to tag objects with the tag type specified by the `tagType` parameter. If you specify 0 for the `tagType` parameter, the `GXGetShapeTags` function searches all tag types.

You can use the `index` and the `count` parameters to specify which tag references of the appropriate type the `GXGetStyleTags` function should return. The `index` parameter indicates the first tag reference to return and the `count` parameter indicates how many tag references to return. The `index` parameter must be greater than 0. The `count` parameter must be greater than 0 or equal to the `gxSelectToEnd` constant (-1), which indicates that all tag references (starting with the tag reference indicated by the `index` parameter) should be returned.

The function result is the number of tag references found that fit the criteria. If you pass a value other than `nil` for the `items` parameter, the `GXGetStyleTags` function returns in it the tag references that were found.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>out_of_memory</code>	
<code>style_is_nil</code>	
<code>index_is_less_than_one</code>	(debugging version)
<code>count_is_less_than_one</code>	(debugging version)

Warnings

<code>index_out_of_range</code>
<code>count_out_of_range</code>

SEE ALSO

Tag objects are introduced in the chapter “Introduction to Objects” in this book. Functions to create and manipulate tags objects, and a list of reserved tag types, are described in the chapter “Tag Objects” in this book.

To change the set of tag references associated with a style, use the `GXSetStyleTags` function, described in the next section.

GXSetStyleTags

You can use the `GXSetStyleTags` function to add, remove, or replace tag objects associated with a style object.

```
void GXSetStyleTags(gxStyle target, long tagType, long index,
                   long oldCount, long newCount,
                   const gxTag items[]);
```

<code>target</code>	A reference to the style object to alter the tag list of.
<code>tagType</code>	The type of tag objects to replace. A value of 0 indicates that you want to replace tags of all types.
<code>index</code>	The (1-based) index of the first tag reference (to a tag object of the appropriate type) to replace.
<code>oldCount</code>	The number of tag references to replace. A value of 0 specifies that you want to insert tag references before the tag reference indicated by the <code>index</code> parameter, rather than replace tag references. A value of -1 (the <code>gxSelectToEnd</code> constant) specifies that all tag references of the requested type, starting with the tag reference indicated by the <code>index</code> parameter, should be replaced.
<code>newCount</code>	The number of tag references to insert. A value of 0 specifies that there are no tag references to insert; the existing tag references that match the criteria you specify in the <code>tagType</code> , <code>index</code> , and <code>oldCount</code> parameters are removed from the source shape's tag list and disposed of.
<code>items</code>	An array of tag references to insert in the tag list.

DESCRIPTION

The `GXSetStyleTags` function allows you add tag references to a style object's tag list, to remove tag references from the list, or to replace tag references in the list with new tag references. In any of these three cases, the `target` parameter specifies the style object to be modified, the `newCount` parameter specifies the number of tag references to add, and the `items` parameter provides the new tag reference.

- n To add tag references, set the `oldCount` parameter to 0. Use the `tagType` and the `index` parameters to specify where to add the new tag references. (For example, if you specify `nil` for the `tagType` parameter and 1 for the `index` parameter, this function inserts the new tag references before the current tag references. If you specify a value other than `nil` for the `tagType` parameter and a value of 2 for the `index` parameter, the function inserts the new tag references before the second tag reference with a tag type matching the `tagType` parameter.)

Style Objects

- n To remove tag references, set the `newCount` parameter to 0 and the `items` parameter to `nil`. You can use the `index` and the `oldCount` parameters to specify which tag references (of the specified type) should be removed. The `index` parameter indicates the first tag reference (of the specified type) to remove and the `oldCount` parameter indicates how many tag references (of the specified type) to remove.
- n To replace tag references, use the `tagType`, `index`, and `oldCount` parameters to indicate which tag references to replace, and use the `newCount` and `items` parameters to specify the new tag references to add. If `newCount` is greater than `oldCount`, the extra tag references are placed immediately adjacent to the last tag reference replaced.

ERRORS, WARNINGS, AND NOTICES

Errors

<code>out_of_memory</code>	
<code>style_is_nil</code>	
<code>tag_is_nil</code>	
<code>parameter_is_nil</code>	(debugging version)
<code>inconsistent_parameters</code>	(debugging version)
<code>parameter_out_of_range</code>	(debugging version)
<code>index_is_less_than_zero</code>	(debugging version)
<code>cannot_dispose_locked_tag</code>	(debugging version)

Warnings

`index_out_of_range`
`count_out_of_range`

Notices (debugging version)

`tag_already_set`

SEE ALSO

Tag objects are introduced in the chapter “Introduction to Objects” in this book. Functions to create and manipulate tags objects, and a list of reserved tag types, are described in the chapter “Tag Objects” in this book.

To examine the set of tag references associated with a style, use the `GXGetStyleTags` function, described in the previous section.

Summary of Style Objects

Constants and Data Types

The style object

```
typedef struct gxPrivateStyleRecord *gxStyle;
```

Functions

Creating and Manipulating Style Objects

```
gxStyle GXNewStyle          (void);  
void GXDisposeStyle        (gxStyle target);  
gxStyle GXCopyToStyle      (gxStyle target, gxStyle source);  
boolean GXEqualStyle       (gxStyle one, gxStyle two);  
gxStyle GXCloneStyle       (gxStyle source);
```

Manipulating Style Object Properties

```
void GXResetStyle          (gxStyle target);  
long GXGetStyleOwners      (gxStyle source);  
long GXGetStyleTags        (gxStyle source, long tagType, long index,  
                             long count, gxTag items[]);  
void GXSetStyleTags        (gxStyle target, long tagType, long index,  
                             long oldCount, long newCount,  
                             const gxTag items[]);
```

Colors and Color-Related Objects

Contents

About Color in QuickDraw GX	4-5
Color Spaces	4-6
Luminance-Based Color Spaces	4-7
RGB-Based Color Spaces	4-9
CMYK Color Spaces	4-14
Universal Color Spaces	4-15
Indexed Color Spaces	4-22
Color Spaces With Alpha Channels	4-24
Color-Component Values, Color Values, and Colors	4-25
Color Conversion and Color Matching	4-26
Color Profiles	4-28
Color-Matching Methods	4-30
When Color Matching Occurs	4-31
About Color Set Objects	4-32
Color Set Properties	4-33
Color Values in a Color Set	4-34
Default Color Sets	4-34
About Color Profile Objects	4-35
Color Profile Properties	4-36
Profile Data	4-36
The Default Color Profile	4-37
Zero-Length Profiles	4-37
Using Colors and Color-Related Objects	4-38
Assigning Colors to Shapes	4-38
Assigning Color Profiles to Colors	4-39

Comparing and Testing Colors	4-40
Checking for Out-of-Gamut Colors	4-40
Checking Colors for Closeness and Color Space	4-40
Predicting Drawing Results	4-41
Converting and Matching Colors	4-41
Creating and Manipulating Color Set and Color Profile Objects	4-42
Creating and Disposing of a Color Set or Color Profile	4-42
Copying, Comparing, and Cloning Color Sets and Color Profiles	4-44
Loading and Unloading Color Sets and Color Profiles	4-45
Manipulating Object Properties of Color Sets and Color Profiles	4-46
Manipulating Owner Counts	4-46
Getting and Setting Tag References	4-47
Manipulating the Colors in a Color Set Object	4-47
Manipulating the Profile Data in a Color Profile Object	4-48
Colors and Color-Related Objects Reference	4-49
Constants and Data Types	4-50
Color-Component Values	4-50
Color Values	4-50
The Color Structure	4-53
Color Packing	4-54
Color Spaces	4-55
The Color Set Object	4-56
The gxSetColor Union	4-56
The Color Profile Object	4-57
Color Functions	4-57
GXCheckColor	4-57
GXGetColorDistance	4-58
GXCombineColor	4-59
GXConvertColor	4-60
Color Set Functions	4-62
Creating and Manipulating Color Set Objects	4-62
GXGetDefaultColorSet	4-62
GXSetDefaultColorSet	4-63
GXNewColorSet	4-64
GXDisposeColorSet	4-65
GXCopyToColorSet	4-66
GXEqualColorSet	4-67
GXCloneColorSet	4-68
Manipulating Color Set Object Properties	4-69
GXGetColorSetOwners	4-69
GXGetColorSetTags	4-70
GXSetColorSetTags	4-71
Retrieving and Replacing Colors in a Color Set	4-73
GXGetColorSet	4-73
GXSetColorSet	4-74
GXGetColorSetParts	4-75
GXSetColorSetParts	4-76

Color Profile Functions	4-78
Creating and Manipulating Color Profile Objects	4-78
GXGetDefaultColorProfile	4-78
GXNewColorProfile	4-79
GXDisposeColorProfile	4-80
GXCopyToColorProfile	4-81
GXEqualColorProfile	4-82
GXCloneColorProfile	4-83
Manipulating Color Profile Object Properties	4-84
GXGetColorProfileOwners	4-84
GXGetColorProfileTags	4-85
GXSetColorProfileTags	4-86
Retrieving and Replacing Profile Information	4-88
GXGetColorProfile	4-88
GXSetColorProfile	4-89
GXLockColorProfile	4-90
GXUnlockColorProfile	4-91
GXGetColorProfileStructure	4-92
Summary of Colors and Color-Related Objects	4-94
Constants and Data Types	4-94
Color Functions	4-98
Color Set Functions	4-98
Color Profile Functions	4-99

Colors and Color-Related Objects

This chapter describes the QuickDraw GX color architecture and the objects and structures with which you manipulate colors. Read this chapter if your application does any color drawing or calculation, or if you create or modify bitmaps or color sets. Read this chapter also if you are creating a calibration program to generate color profiles.

Before reading this chapter, you should be familiar with the information in the chapter “Introduction to QuickDraw GX” in this book. You should also be familiar with shape objects, as discussed in the chapter “Shape Objects” in this book.

This chapter constitutes the complete discussion of color for QuickDraw GX. Unlike for shape objects and style objects, there is no additional discussion of color-related objects in other books. However, additional information relevant to color is in the chapter “Ink Objects” in this book.

QuickDraw GX uses color-matching methods provided by the Macintosh ColorSync Utilities. For information on ColorSync, its color-matching capabilities, and the structure of the color profiles it uses, see the ColorSync chapter of *Inside Macintosh: Advanced Color Imaging* and the Component Manager chapter of *Inside Macintosh: More Macintosh Toolbox*.

For general information on color theory and color spaces, you may also want to read other books such as these: *Measuring Color*, by R.W.G. Hunt, John Wiley & Sons, New York, 1991; *Illumination and Color in Computer Generated Imagery*, by Roy Hall, Springer-Verlag, New York, 1989; and *Computer Graphics: Principles and Practice*, by J. Foley, A. van Dam, S. Feiner, and J. Hughes, Addison-Wesley, Reading, 1990.

This chapter introduces how color is represented in QuickDraw GX, and describes color set objects and color profile objects and their properties. It then shows how to use QuickDraw GX functions to

- n assign colors to shapes
- n compare, test, and convert colors
- n automatically use the color-matching capabilities of QuickDraw GX
- n create and manipulate color profiles for imaging devices
- n manipulate the colors of a bitmap that uses indexed colors

About Color in QuickDraw GX

In QuickDraw GX, color information about a shape is kept in the ink object associated with the shape object. A shape’s ink object describes both the color of the shape and the transfer mode with which the shape is drawn. Ink objects are described in the chapter “Ink Objects” in this book; colors are described in this chapter.

QuickDraw GX has a powerful, device-independent method for representing color in many different formats. Conversion among the formats is simple and direct, and in many cases automatic. QuickDraw GX also provides automatic manipulation of device-specific colors so that colors match consistently when scanned from or drawn to many different imaging devices.

This section describes how color is represented and how you can manipulate color information. It presents the information in this order:

- n Colors are numerical values that make sense only in terms of specific color spaces. Color spaces are described first, under “Color Spaces” (next section).
- n The mathematical values used by each color space are combined with other information to make a color structure. How color values relate to the color structure is described second, under “Color-Component Values, Color Values, and Colors” beginning on page 4-25.
- n Colors in a given color space or produced with a given input or display device commonly must be converted to another color space or matched to the color capabilities of another device. How QuickDraw GX accomplishes that task is described third, under “Color Conversion and Color Matching” beginning on page 4-26.

Color Spaces

A **color space** specifies how color information is represented. It defines a one-, two-, three-, or four-dimensional space whose dimensions, or **components**, represent intensity values. For example, RGB space is a three-dimensional color space whose components are the red, green, and blue intensities that make up a given color. Visually, these spaces are often represented by various solid shapes, such as cubes, cones, or polyhedra. See, for example, Color Plate 4 at the front of this book.

QuickDraw GX directly supports 28 different color spaces, to give you the convenience of working in whatever kinds of color data most suits your needs. The QuickDraw GX color spaces fall into several groups, or **base families**. They are

- n luminance-based color spaces, used for grayscale display and printing
- n RGB-based color spaces, used mainly for color video display
- n CMYK-based color spaces, used mainly for color printing
- n universal color spaces, used mainly for device-independent color measurements

All color spaces within a base family differ only in details of storage format or else are related to each other by very simple mathematical formulas. Conversion of color across base families is more complex, as described in the section “Color Conversion and Color Matching” beginning on page 4-26.

Within a base family, some of the differences among color spaces relate to their **packing**, the number of bits used to store each color component. For example, RGB colors might be stored with 5, 8, or 16 bits per component. Each storage format is a different color space. Internally, QuickDraw GX always converts colors so that each component has 16 bits; thus you can think of the 16-bit-per-component color spaces as the fundamental ones in each base family, and those with smaller storage spaces as packed (storage-compressed) versions.

Some QuickDraw GX color spaces have an alpha channel, an additional component that measures opacity or transparency. Alpha channels are described in the section “Color Spaces With Alpha Channels” beginning on page 4-24.

QuickDraw GX also supports a derived color space—indexed color space—in which colors are indirectly specified, using values that are indexed positions in a list. The colors in that list, however, must still belong to one of the base-family color spaces.

The `gxColorSpaces` enumeration, shown on page 4-55, lists the color spaces directly supported by QuickDraw GX. Each color space has its own format for representing color information. The rest of this section discusses those color spaces and their formats.

Luminance-Based Color Spaces

Luminance is a scale of lightness. Luminance-based color spaces, or gray spaces, typically have a single component, ranging from black to white, as shown in Figure 4-1. Luminance-based color spaces are used for black-and white and grayscale display and printing.

Figure 4-1 Luminance color space



A color is converted into luminance by evaluating its overall lightness. The luminance of a color expressed in RGB (see “RGB-Based Color Spaces” beginning on page 4-9), for example, can be calculated approximately with this formula:

```
luminance = 0.30 * red + 0.59 * green + 0.11 * blue;
```

(QuickDraw GX provides a function for converting colors among different color spaces.)

The luminance-based color spaces supported by QuickDraw GX (and defined in the `gxColorSpaces` enumeration) are `gxGraySpace` and `gxGrayASpace`. The *A* in `gxGrayASpace` stands for a second component called an *alpha channel*; see the section “Color Spaces With Alpha Channels” beginning on page 4-24 for more information.

Table 4-1 describes details of the storage formats for `gxGraySpace` and `gxGrayASpace`. In each of these spaces, the luminance is specified by a single number whose range varies from 0 to 65,535. The color black has a luminance value of 0, regardless of the color space.

Table 4-1 Luminance-based color spaces supported by QuickDraw GX

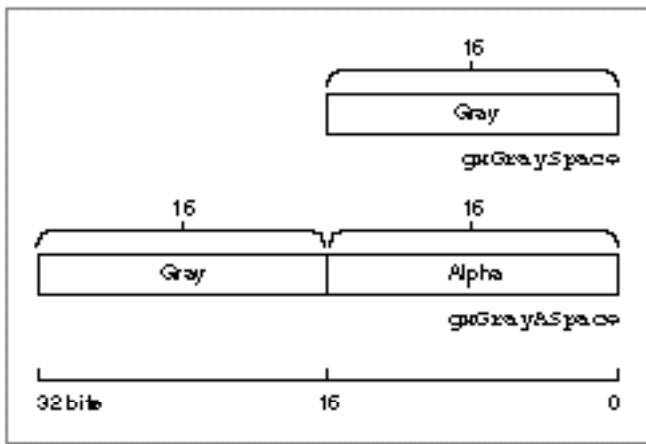
Constant	Enumeration Value	Explanation
<code>gxGraySpace</code>	<code>0x000A</code>	16 bits per component (gray only); component values range from 0 to 0xFFFF. Total storage size for each color value: 16 bits.
<code>gxGrayASpace</code>	<code>0x008A</code>	16 bits per component (gray and alpha); component values range from 0 to 0xFFFF. Total storage size for each color value: 32 bits. Alpha channels are described on page 4-24.

Figure 4-2 is a visual representation of the storage formats for the luminance-based color spaces.

Note

This figure and all subsequent storage-format figures in this chapter assume that data storage is “big-endian,” that is, that lower addresses correspond to higher-order bytes in a word or long word value. For processors whose storage model is different, the elements of the figures would be in a different order. These figures are presented for illustrative purposes only, and are not intended to specify details of storage order. u

Figure 4-2 Storage formats for luminance-based color spaces



QuickDraw GX does not support an 8-bit luminance-based color space because such a color space can be more conveniently represented as an indexed color space with a color set. Indexed color space is described in the section “Indexed Color Spaces” beginning on page 4-22; color sets are described in the section “When Color Matching Occurs” beginning on page 4-31.

RGB-Based Color Spaces

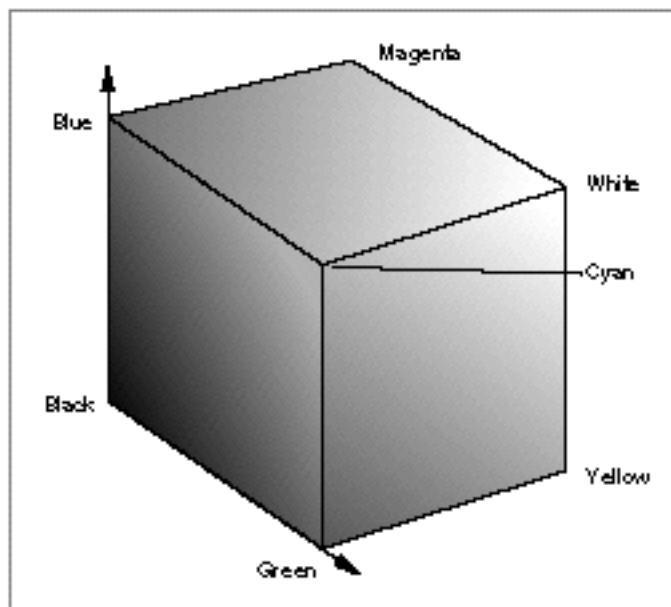
RGB-based color spaces are the most commonly used color spaces in computer graphics, primarily because they are directly supported by most color monitors. The groups of color spaces within the RGB base family include

- n RGB spaces
- n HSV and HLS spaces

RGB Spaces

Any color expressed in **RGB space** is some mixture of three primary colors red, green, and blue. Most RGB-based color spaces can be visualized as a cube, as in Figure 4-3, with corners of black, the three primaries (red, green, and blue), the three secondaries (cyan, magenta, and yellow), and white. See, for example, Figure 4-3; see also Color Plate 4 at the front of this book.

Figure 4-3 RGB color space

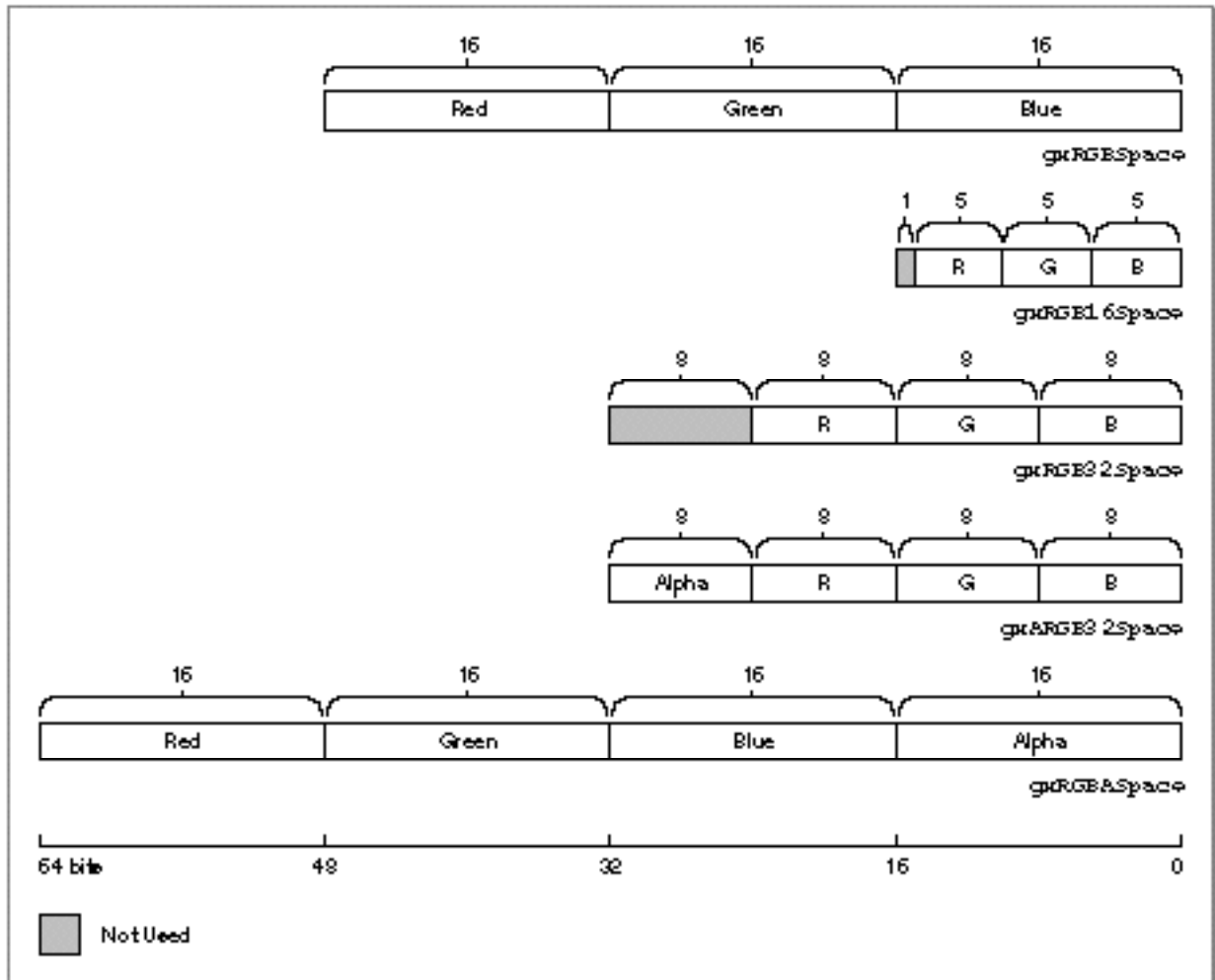


Colors and Color-Related Objects

The RGB color spaces supported by QuickDraw GX (and defined in the `gxColorSpaces` enumeration) are `gxRGBSpace`, `gxRGB16Space`, `gxRGB32Space`, `gxRGBASpace`, and `gxARGB32Space`. See Table 4-2 and Figure 4-4 for storage-format details. In each of these spaces, a color value is represented by three or four color components, representing red, green, blue (and in some cases alpha); each component can vary in the number of bits used for its storage. The color black is represented by component values of 0 in the red, green, and blue components.

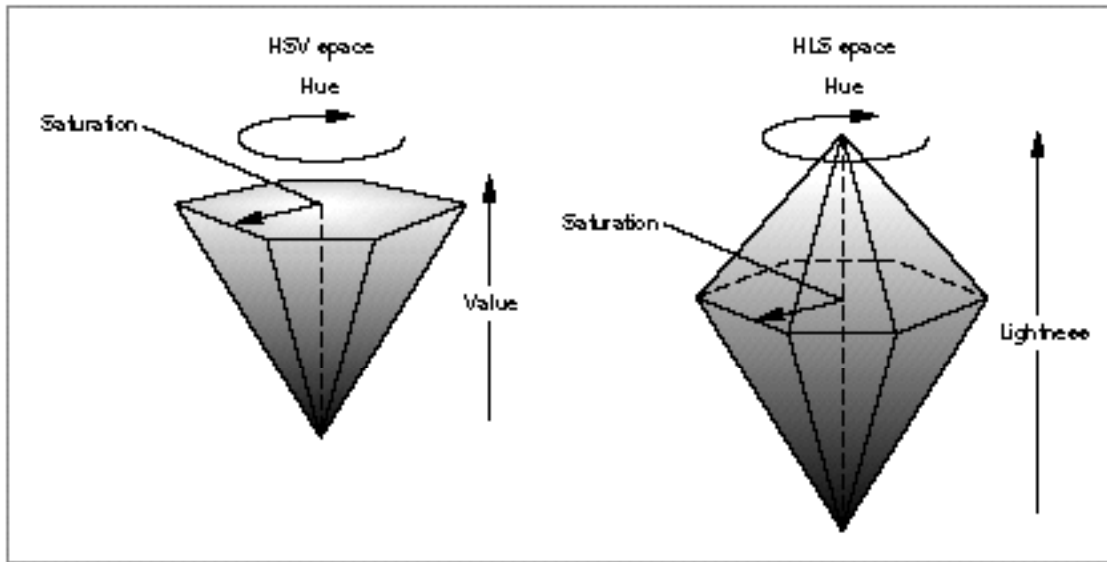
Table 4-2 RGB color spaces supported by QuickDraw GX

Constant	Enumeration Value	Explanation
<code>gxRGBSpace</code>	<code>0x0001</code>	16 bits per component (red, green, and blue); component values range from 0 to 0xFFFF. Total storage size for each color value: 48 bits.
<code>gxRGB16Space</code>	<code>0x0501</code>	5 bits per component (red, green, and blue); component values range from 0 to 0x1F. Total storage size for each color value: 16 bits (bit 15 is not used).
<code>gxRGB32Space</code>	<code>0x0801</code>	8 bits per component (red, green, and blue); component values range from 0 to 0xFF. Total storage size for each color value: 32 bits (bits 24–31 are not used).
<code>gxARGB32Space</code>	<code>0x1881</code>	8 bits per component (red, green, blue, and alpha); component values range from 0 to 0xFF. Total storage size for each color value: 32 bits. Alpha channels are described on page 4-24.
<code>gxRGBASpace</code>	<code>0x0081</code>	16 bits per component (red, green, blue, and alpha); component values range from 0 to 0xFFFF. Total storage size for each color value: 64 bits. Alpha channels are described on page 4-24.

Figure 4-4 Storage formats for RGB color spaces

HSV and HLS Color Spaces

HSV space and **HLS space** are transformations of RGB space that allow colors to be described in terms more natural to an artist. The name *HSV* stands for *hue*, *saturation*, and *value*, and *HLS* stands for *hue*, *lightness*, and *saturation*. The two spaces can be thought of as being single and double cones, as shown in Figure 4-5. (See also Color Plate 4 at the front of this book for a slightly different representation of these color spaces.)

Figure 4-5 HSV color space and HLS color space

The components in HLS space are analogous, but not completely identical, to the components in HSV space:

- n The hue component in both color spaces is an angular measurement, analogous to position around a color wheel. A hue value of 0 indicates the color red; the color green is at a value corresponding to 120°, and the color blue is at a value corresponding to 240°. Horizontal planes through the cones in Figure 4-5 are hexagons; the primaries and secondaries (red, yellow, green, cyan, blue, and magenta) occur at the vertices of the hexagons.
- n The saturation component in both color spaces describes color intensity. A saturation value of 0 (in the middle of a hexagon) means that the color is “colorless” (gray); a saturation value at the maximum (at the outer edge of a hexagon) means that the color is at maximum “colorfulness” for that hue angle and brightness.
- n The value component (in HSV space) and the lightness component (in HLS space) describe brightness or luminance. In both color spaces, a value of 0 represents black. In HSV space, a maximum value for value means that the color is at its brightest. In HLS space, a maximum value for lightness means that the color is white, regardless of the current values of the hue and saturation components. The brightest, most intense color in HLS space occurs at a lightness value of exactly half the maximum.

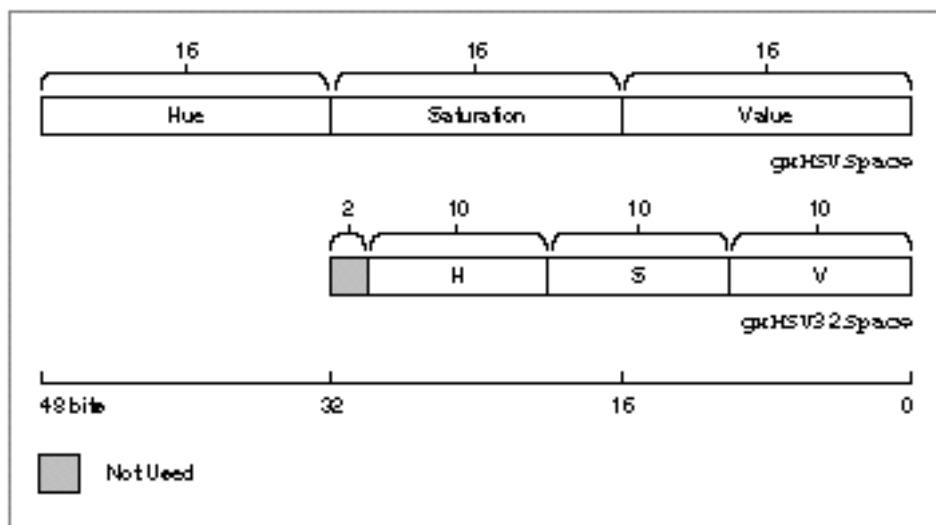
The HLS and HSV color spaces supported by QuickDraw GX (and defined in the `gxColorSpaces` enumeration) are `gxHSVSpace`, `gxHLSSpace`, `gxHSV32Space`, and `gxHLS32Space`. See Table 4-3 and Figure 4-6 for details of storage format.

Table 4-3 HSV and HLS color spaces supported by QuickDraw GX

Constant	Enumeration Value	Explanation
<code>gxHSVSpace</code>	<code>0x0003</code>	16 bits per component (hue, saturation, and value); component values range from 0 to 0xFFFF. Total storage size for each color value: 48 bits.
<code>gxHLSSpace</code>	<code>0x0004</code>	16 bits per component (hue, lightness, and saturation); component values range from 0 to 0xFFFF. Total storage size for each color value: 48 bits.
<code>gxHSV32Space</code>	<code>0x0A03</code>	10 bits per component (hue, saturation, and value); component values range from 0 to 0x3FF. Total storage size for each color value: 32 bits (bits 30–31 are not used).
<code>gxHLS32Space</code>	<code>0x0A04</code>	10 bits per component (hue, lightness, and saturation); component values range from 0 to 0x3FF. Total storage size for each color value: 32 bits (bits 30–31 are not used).

Figure 4-6 shows storage formats for the supported HSV color spaces. Formats for the HLS spaces are identical.

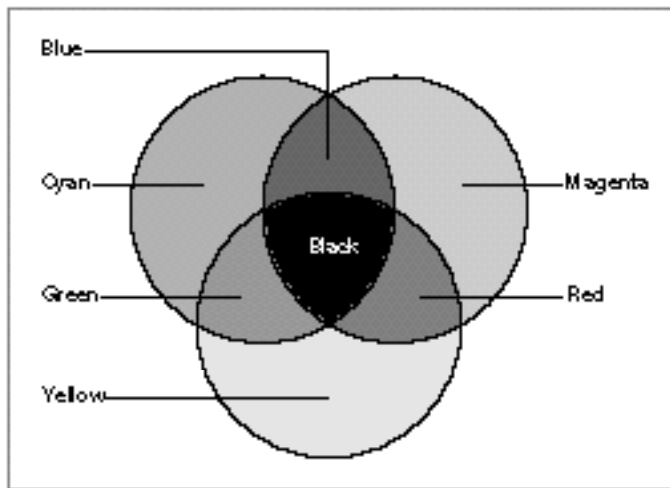
Figure 4-6 Storage formats for HSV color spaces



CMYK Color Spaces

CMYK space is a color space that models the way ink builds up in printing. The name *CMYK* refers to cyan, magenta, yellow, and black. Cyan, magenta, and yellow are the three primary colors in this color space, and red, green, and blue are the three secondaries. Theoretically black is not needed. However, when full-saturation cyan, magenta, and yellow inks are mixed equally on paper, the result is usually a dark brown, rather than black. Therefore, black ink is overprinted in darker areas to give a better appearance. Figure 4-7 shows how the primary colors in CMYK space mix to form other colors. (See also Color Plate 4 at the front of this book.)

Figure 4-7 Colors in CMYK color space



Theoretically, the relation between RGB values and CMY values in CMYK space is quite simple:

```
Cyan      = 1.0 - red;
Magenta   = 1.0 - green;
Yellow    = 1.0 - blue;
```

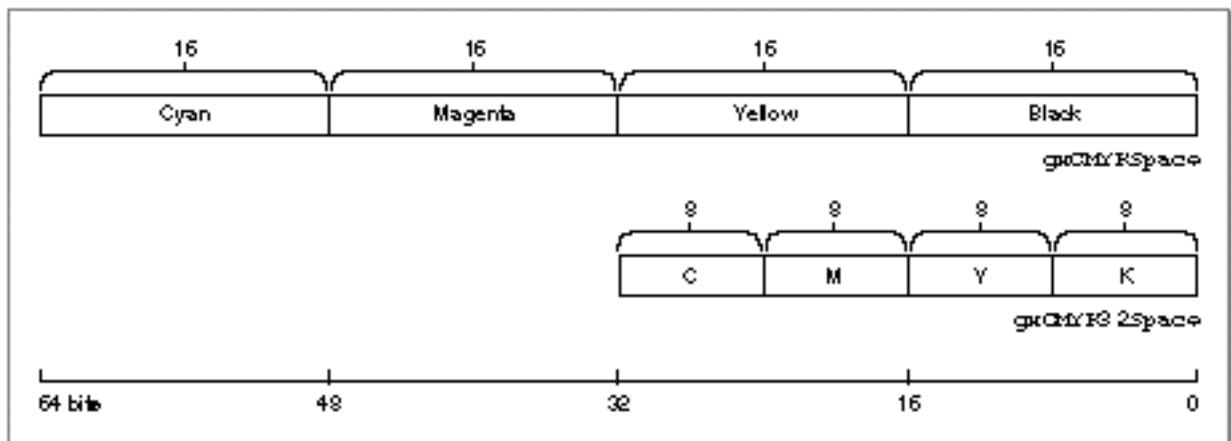
(where red, green, and blue intensities are expressed as fractional values varying from 0 to 1). In reality, the process of deriving the cyan, magenta, yellow, and black values from a color expressed in RGB space is complex, involving device-specific, ink-specific, and even paper-specific calculations of the amount of black to add in dark areas (**black generation**), and the amount of other ink to remove (**undercolor removal**) where black is to be printed. QuickDraw GX performs those calculations for you when converting among color spaces, commonly using color profiles as described in the section “Color Profiles” beginning on page 4-28.

The CMYK color spaces supported by QuickDraw GX (and defined in the `gxColorSpaces` enumeration) are `gxCMYKSpace` and `gxCMYK32Space`. See Table 4-4 and Figure 4-8 for details of storage format.

Table 4-4 CMYK color spaces supported by QuickDraw GX

Constant	Enumeration Value	Explanation
<code>gxCMYKSpace</code>	<code>0x0002</code>	16 bits per component (cyan, magenta, yellow, and black); component values range from 0 to 0xFFFF. Total storage size for each color value: 64 bits.
<code>gxCMYK32Space</code>	<code>0x0802</code>	8 bits per component (cyan, magenta, yellow, and black); component values range from 0 to 0xFF. Total storage size for each color value: 64 bits.

Figure 4-8 Storage formats for CMYK color spaces



Universal Color Spaces

Some color spaces allow color to be expressed in a device-independent way. Whereas RGB colors vary with monitor characteristics, and CMYK colors vary with printer and paper characteristics, *universal colors* are meant to be true representations of colors as perceived by the human eye. These color representations, called **universal color spaces**, result from work carried out in 1931 by the Commission Internationale d’Eclairage (CIE), and for that reason are also called *CIE-based color spaces*.

In addition, broadcast-video color space (YIQ) is based on device-independent color characteristics, in that its colors are measured in terms of a standard device. It is therefore considered universal and is discussed in this section.

XYZ Space

There are several CIE-based color spaces, but all are derived from the fundamental **XYZ space**. The XYZ space allows colors to be expressed as a mixture of the three **tristimulus values** X, Y, and Z. The term *tristimulus* comes from the fact that color perception results from the retina of the eye responding to three types of stimuli. After experimentation, the CIE set up a hypothetical set of primaries, XYZ, that correspond to the way the eye's retina behaves.

The CIE defined the primaries so that all visible light maps into a positive mixture of X, Y, and Z, and so that Y correlates approximately to the apparent lightness of a color. Generally, the mixtures of X, Y, and Z components used to describe a color are expressed as percentages ranging from 0% up to, in some cases, just over 100%.

Other universal color spaces based on XYZ space are used primarily to relate some particular aspect of color or some perceptual color difference to XYZ values.

Yxy Space

Yxy space expresses the XYZ values in terms of x and y **chromaticity** coordinates, somewhat analogous to the hue and saturation coordinates of HSV space. The coordinates are shown in the following formulas, used to convert XYZ into Yxy:

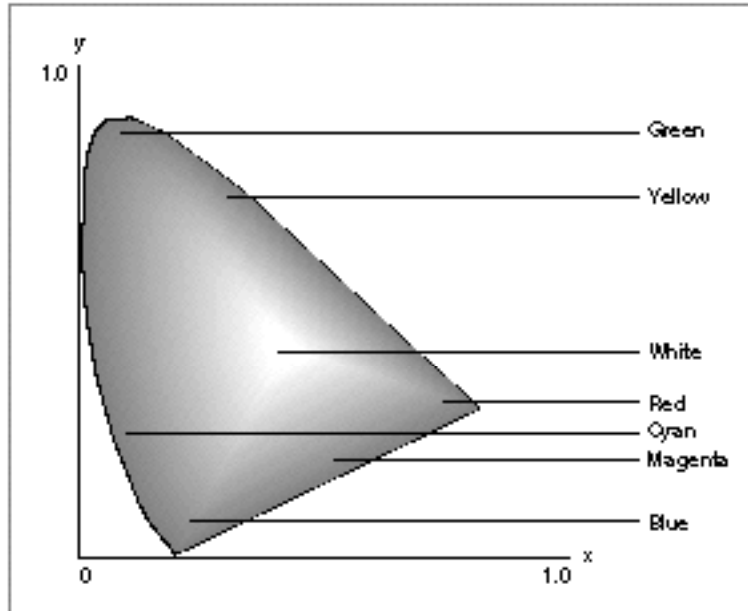
$$Y = Y$$

$$x = X / (X+Y+Z)$$

$$y = Y / (X+Y+Z)$$

Note that the Z tristimulus value is incorporated into the new coordinates, and does not appear by itself. Since Y still correlates to the lightness of a color, the other aspects of the color are found in the chromaticity coordinates x and y. This allows color variation in Yxy space to be plotted on a two-dimensional diagram. Figure 4-9 shows the layout of colors in the x and y plane of Yxy space. Color Plate 4 at the front of this book shows the same plot in color.

Figure 4-9 Yxy chromaticities



L*u*v* Space and L*a*b* Space

One problem with representing colors using the XYZ and Yxy color spaces is that they are perceptually nonlinear: it is not possible to accurately evaluate the perceptual closeness of colors based on their relative positions in XYZ or Yxy space. Colors that are close together in Yxy space may seem very different to observers, and colors that seem very similar to observers may be widely separated in Yxy space.

L*u*v* space is a nonlinear transformation of XYZ space in order to create a perceptually linear color space. **L*a*b* space** is a nonlinear transformation (a third-order approximation) of the Munsell color-notation system (not described here). Both are designed to match perceived color difference with quantitative distance in color space.

Both L*u*v* space and L*a*b* space represent colors relative to a **reference white point**, which is a specific definition of what is considered white light, represented in terms of XYZ space, and usually based on the whitest light that can be generated by a given device. (In that sense L*u*v* and L*a*b* are not completely device independent; two numerically equal colors are truly identical only if they were measured relative to the same white point.)

Measuring colors in relation to a white point allows for color measurement under a variety of illuminations. The luminance of the white point of the QuickDraw GX default color profile matches the luminance of the white point on the Apple 13-inch color monitor. Color profiles are described in the section “Color Conversion and Color Matching” beginning on page 4-26.

A primary benefit of using L*u*v* space and L*a*b* space is that the perceived difference between any two colors is proportional to the geometric distance in the color space between their color values. For applications where closeness of color needs to be quantified, such as in colorimetry, gemstone evaluation, or dye matching, use of L*u*v* space or L*a*b* space is common.

The formulas for transforming an XYZ color into an L*u*v* color are

```
if (Y/Yn > 0.008856)
    L = 116.0 * (Y / Yn)1/3 - 16.0;
else
    L = 903.3 * (Y / Yn);
u = 13.0 * L * (u' - u'n);
v = 13.0 * L * (v' - v'n);
```

where

```
u' = 4 * x / (X + 15*Y + 3*Z);
v' = 9 * y / (X + 15*Y + 3*Z);
```

and u'n, v'n, and Yn are the u', v', and Y values for the reference white point.

Similarly, the formulas for transforming an XYZ color into an L*a*b* color are

```
if (Y/Yn > 0.008856)
    L = 116.0 * (Y / Yn)1/3 - 16.0;
else
    L = 903.3 * (Y / Yn)
a = 500.0 * ( (X / Xn)1/3 - (Y / Yn)1/3 );
b = 500.0 * ( (Y / Yn)1/3 - (Z / Zn)1/3 );
```

where Xn, Yn, and Zn are the XYZ values for the reference white point.

Formats for XYZ-Based Color Spaces

The universal color spaces supported by QuickDraw GX (and defined in the gxColorSpaces enumeration) are gxYXYSpace, gxXYZSpace, gxLUVSpace, gxLABSpace, gxYXY32Space, gxXYZ32Space, gxLUV32Space, and gxLAB32Space. See Table 4-5 and Figure 4-10 for details of storage format. Note that the ranges of values for the components differ significantly among the different color spaces.

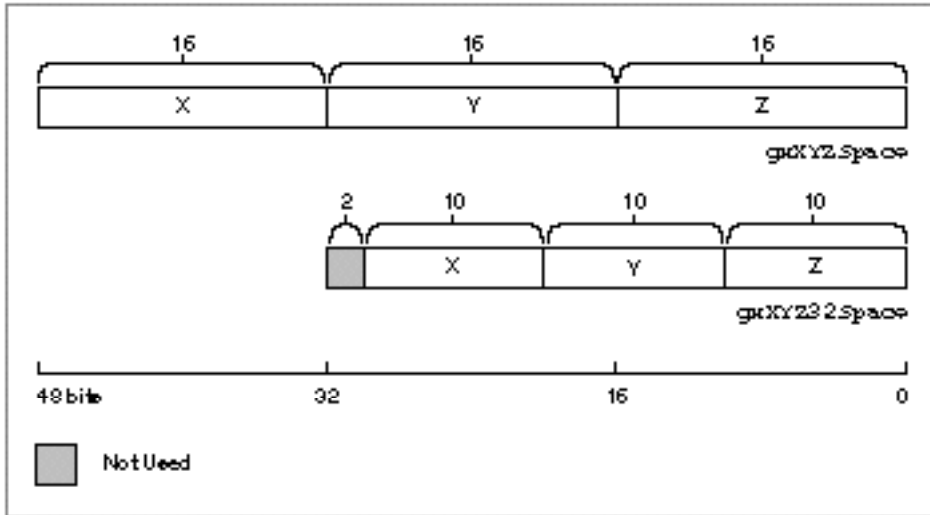
Figure 4-10 shows storage formats for the supported XYZ color spaces. Formats for the Yxy, L*u*v*, and L*a*b* spaces are identical.

Table 4-5 Universal color spaces supported by QuickDraw GX

Constant	Enumeration Value	Explanation
gxYXYSpace	0x0005	16 bits per component (Y, x, and y); component values range from 0 (0%) to 0xFFFF (100%). Total storage size for each color value: 48 bits.
gxXYZSpace	0x0006	16 bits per component (X, Y, and Z). Component values range from 0 (0%) to 0xFFFF (200%; a value of 0x8000 represents 100%). Total storage size for each color value: 48 bits.
gxLUVSpace	0x0007	16 bits per component (L*, u*, and v*). The L* component values range from 0 (0%) to 0xFFFF (100% of white-point luminance). The u* and v* component values range from 0 (-1) to 0xFFFF (+1). Total storage size for each color value: 48 bits.
gxLABSpace	0x0008	16 bits per component (L*, a*, and b*). The L* component values range from 0 (0%) to 0xFFFF (100% of white-point luminance). The a* and b* component values range from 0 (-1) to 0xFFFF (+1). Total storage size for each color value: 48 bits.
gxYXY32Space	0x0A05	10 bits per component (Y, x, and y); component values range from 0 (0%) to 0x3FF (100%). Total storage size for each color value: 32 bits (bits 30 and 31 not used).
gxXYZ32Space	0x0A06	10 bits per component (X, Y, and Z). Component values range from 0 (0%) to 0x3FF (200%; a value of 0x200 represents 100%). Total storage size for each color value: 32 bits (bits 30 and 31 not used).
gxLUV32Space	0x0A07	10 bits per component (L*, u*, and v*). The L* component values range from 0 (0%) to 0x3FF (100% of white-point luminance). The u* and v* component values range from 0 (-1) to 0x3FF (+1). Total storage size for each color value: 32 bits (bits 30 and 31 not used).
gxLAB32Space	0x0A08	10 bits per component (L*, a*, and b*). The L* component values range from 0 (0%) to 0x3FF (100% of white-point luminance). The a* and b* component values range from 0 (-1) to 0x3FF (+1). Total storage size for each color value: 32 bits (bits 30 and 31 not used).

NOTE Because u*, v*, a*, and b* are normally signed numbers between 1.0 and -1.0, you can convert them into shorts as follows:

```
anUnsignedshort = ((aFloat + 1.0)/2) * 65535.0;
```

Figure 4-10 Storage formats for XYZ color spaces

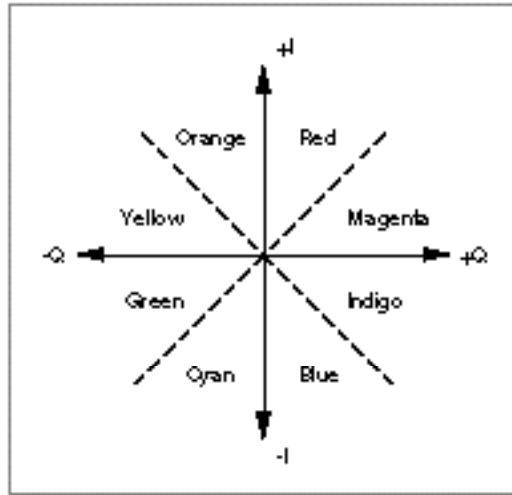
Video Color Spaces

YIQ space is sometimes called *video color space*. It is based on the way a specific kind of RGB data is broken down for color television transmission. The three dimensions that describe these color spaces are Y, I, and Q, in which Y represents luminance and the other two components carry color information.

Because the Y channel represents luminance it can be used alone; the Y channel is the only channel used in black and white television. The I and Q channels are called *color difference channels*: the Y channel is split between them. The notations "I" and "Q" stand for "in phase" and "in quadrature," respectively, referring to the method by which all of the channels are combined into a signal for broadcast.

QuickDraw GX also defines NTSC and PAL color spaces. NTSC space corresponds to the color encoding used for color broadcasting in the United States, whereas PAL space corresponds to the color encoding used in Europe. NTSC and PAL have different screen resolutions, frequencies, and are otherwise incompatible, but in terms of how color values are calculated, NTSC space and PAL space are both identical to YIQ space.

In YIQ space, the Y component can vary from 0 (black) to its maximum value (full luminance). I and Q are normally signed values, so they are centered around 0. Figure 4-11 illustrates how colors map into the I and Q dimensions of YIQ space.

Figure 4-11 The I and Q axes in YIQ color space

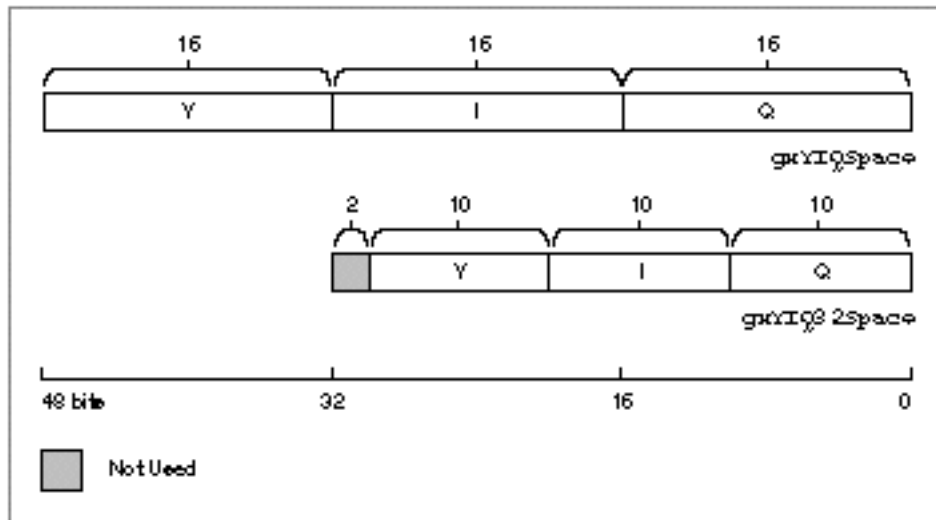
The video color spaces supported by QuickDraw GX (and defined in the `gxColorSpaces` enumeration) are `gxYIQSpace`, `gxNTSCSpace`, `gxPALSpace`, `gxYIQ32Space`, `gxNTSC32Space`, and `gxPAL32Space`. See Table 4-6 and Figure 4-12 for details of storage format. In each of these spaces, a color value is represented by Y, I, and Q color components.

Table 4-6 Video color spaces supported by QuickDraw GX

Constant	Enumeration Value	Explanation
<code>gxYIQSpace</code>	<code>0x0009</code>	16 bits per component (Y, I, and Q); Y-component values range from 0 to <code>0xFFFF</code> ; I- and Q-component values range from <code>-0x7FFF</code> to <code>+0x7FFF</code> . Total storage size for each color value: 48 bits.
<code>gxNTSCSpace</code>	<code>0x0009</code>	(same as <code>gxYIQSpace</code>)
<code>gxPALSpace</code>	<code>0x0009</code>	(same as <code>gxYIQSpace</code>)
<code>gxYIQ32Space</code>	<code>0x0A09</code>	10 bits per component (Y, I, and Q); Y-component values range from 0 to <code>0x3FF</code> ; I- and Q-component values range from <code>-0x1FF</code> to <code>+0x1FF</code> . Total storage size for each color value: 32 bits (bits 30 and 31 are not used).
<code>gxNTSC32Space</code>	<code>0x0A09</code>	(same as <code>gxYIQ32Space</code>)
<code>gxPAL32Space</code>	<code>0x0A09</code>	(same as <code>gxYIQ32Space</code>)

Figure 4-12 shows storage formats for the supported YIQ color spaces. Formats for the NTSC and PAL spaces are identical.

Figure 4-12 Storage formats for YIQ color spaces



You can find more information on the theories of color and the various color spaces in the following publications:

Measuring Color, by R.W.G. Hunt, John Wiley & Sons, New York, 1987.

Illumination and Color in Computer Generated Imagery, by Roy Hall, Springer-Verlag, New York, 1989.

Indexed Color Spaces

In situations where you use only a limited number of colors, it can be impractical or impossible to specify colors directly. For example, if you have a bitmap with only a few bits per pixel (1, 2, 4 or 8 for QuickDraw GX), each pixel is too small to contain a complete color specification; its color must be specified as an index into a list or table of color values. If you are using spot colors in printing or pen colors in plotting, it can be simpler and more precise to specify each color as an index into a list instead of an actual color value. Also, if you want to restrict the user to drawing with a specific set of colors, you can put them in a list and specify them by index.

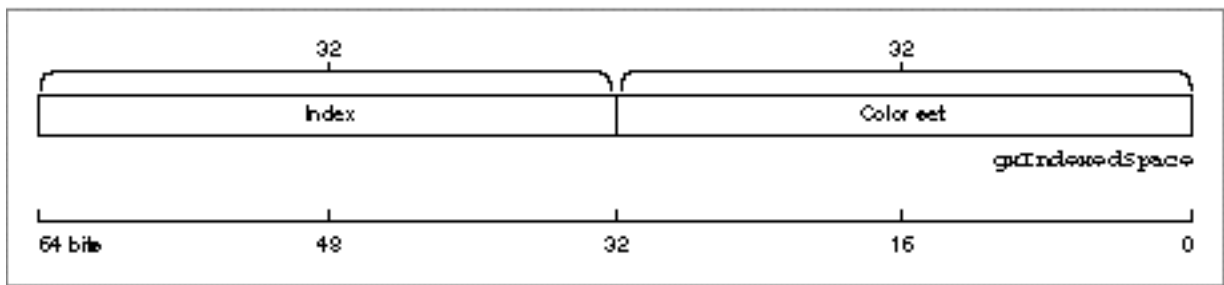
Indexed space is the color space you use when drawing with indirectly specified colors. An indexed color value (a color specification in indexed color space) consists of an index value and a reference to a color set object. The color set contains a list of color values and a specification of the color space for those color values; the index value specifies which color to use from the list. Color values are defined in the section “Color-Component Values, Color Values, and Colors” beginning on page 4-25. Color set objects are described in the section “About Color Set Objects” beginning on page 4-32.

QuickDraw GX supports the single indexed color space format `gxIndexedSpace` (defined in the `gxColorSpaces` enumeration). See Table 4-7 and Figure 4-4 for details of storage format. Although there is a single format for indexed color space, you can create any number of unique indexed color spaces, using different sets of colors from any of the defined color spaces.

Table 4-7 Indexed color space supported by QuickDraw GX

Constant	Enumeration Value	Explanation
<code>gxIndexedSpace</code>	<code>0x000B</code>	Indicates that the color value is a (1-based) index into the referenced color set. Total storage size for each color value: 64 bits.

Figure 4-13 Storage format for indexed color space



Color spaces and bitmaps

Bitmaps commonly use indexed color space, but if pixel size is large enough a bitmap can specify colors directly in any color space. These are the restrictions on the use of color spaces with bitmaps:

- n Bitmaps with 1, 2, 4, or 8 bits per pixel must use `gxIndexedSpace`.
- n Bitmaps with 16 bits per pixel can use `gxRGB16Space`. They cannot use `gxIndexedSpace`.
- n Bitmaps with 32 bits per pixel can use `gxRGB32Space`, `gxARGB32Space`, `gxCMYK32Space`, `gxHSV32Space`, `gxHLS32Space`, `gxYXY32Space`, `gxXYZ32Space`, `gxLUV32Space`, `gxLAB32Space`, `gxYIQ32Space`, `gxNTSC32Space`, or `gxPAL32Space`—that is, all defined 32-bit color spaces. They cannot use `gxIndexedSpace`.
- n Hardware devices that have 24 bits of physical memory per pixel can support `gxRGB32Space`. Hardware devices that have 32 bits of physical memory per pixel can support `gxRGB32Space` plus all the other defined 32-bit color spaces.

Bitmaps are described further in the bitmap shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*. u

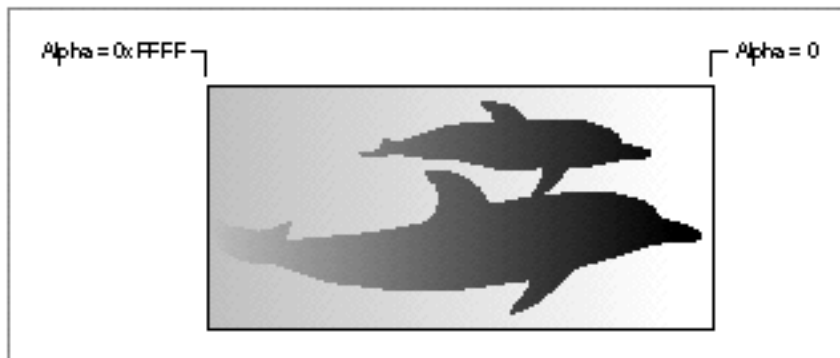
Color Spaces With Alpha Channels

QuickDraw GX supports the use of an alpha channel in one luminance-based color space (`gxGrayASpace`) and two RGB color spaces (`gxRGBASpace` and `gxARGB32Space`). An **alpha channel** is a component in a color space whose value typically determines the opacity of the color expressed by the other components. An alpha-channel value of 0 in a color means that the color is completely transparent, and a maximum value means that the color is completely opaque. A value in between means that the color is partially transparent.

How transparency is handled in drawing depends on the transfer mode used when the color is drawn. (Transfer modes are discussed in the chapter “Ink Objects” in this book.) Typically, however, transparency in a color being drawn—the source color—means that the existing color at the location where drawing occurs—the destination color—shows through. Where the source is completely opaque, the destination is completely covered and is invisible; where the source is completely transparent, the destination shows through unchanged and the source is invisible.

Figure 4-14 shows an example in which a uniform gray image (in `gxGrayASpace`) is drawn over a black-and-white image. The gray color of the source is uniform across the rectangle, but the alpha-channel value decreases from `0xFFFF` on the left to 0 on the right. As the alpha value decreases rightward, more and more of the destination color shows through. (Color Plate 2 at the front of this book shows a similar drawing example in color.)

Figure 4-14 Showing color transparency with an alpha channel



For more information on using alpha channels to achieve particular drawing effects, see the chapter “Ink Objects” in this book.

Color-Component Values, Color Values, and Colors

Each of the color spaces described in this chapter requires one or more numeric values in a particular format to specify a color. This section describes the data types and structures with which QuickDraw GX describes colors in its color spaces.

Each dimension, or component, in a color space has a **color-component value**. In the fundamental, unpacked QuickDraw GX color spaces—those with 16 bits per component—each color-component value is of type `gxColorValue`:

```
typedef unsigned short gxColorValue;
```

A color-component value can vary from 0 to 65,535 (0xFFFF), although the numerical interpretation of that range is different for different color spaces, as has been noted in Table 4-1 through Table 4-7. In most cases, color-component intensities are interpreted numerically as varying between 0 and 1.0; for that reason, QuickDraw GX provides the constant `gxColorValue1` to represent 0xFFFF.

Depending on the color space used, one, two, three, or four color-component values combine to make a **color value**. A color value is a structure; it is the complete specification of a color in a given color space. QuickDraw GX supports 13 color-value formats, representing the fundamental 16-bits-per-component color spaces; all color operations in memory use one of those formats. The color-value formats are described in the section “Color Values” beginning on page 4-50. For example, an RGB color value has this format:

```
struct gxRGBColor{
    gxColorValue    red;
    gxColorValue    green;
    gxColorValue    blue;
};
```

This is exactly the storage format for colors in `gxRGBSpace`. However, colors stored in `gxRGB16Space` or `gxRGB32Space` have a packed storage format, and need to be converted to `gxRGBColor` format when they are used. QuickDraw GX takes care of this for you; as far as your application is concerned, you can always manipulate colors in the color space you have specified.

Colors and Color-Related Objects

A color value plus a specification of the color space it belongs to (plus an optional reference to a color profile to use for color matching) constitute a color in QuickDraw GX. A color is defined by the `gxColor` structure:

```
struct gxColor{
    gxColorSpace          space;
    gxColorProfile        profile;
    union {
        struct gxCMYKColor    cmyk;
        struct gxRGBColor     rgb;
        struct gxRGBAColor    rgba;
        struct gxHSVColor     hsv;
        struct gxHLSColor     hls;
        struct gxXYZColor     xyz;
        struct gxYXYColor     yxy;
        struct gxLUVColor     luv;
        struct gxLABColor     lab;
        struct gxYIQColor     yiq;
        gxColorValue          gray;
        struct gxGrayAColor    graya;
        unsigned short        pixel16;
        unsigned long         pixel32;
        struct gxIndexedColor indexed;
        gxColorValue          component[4];
    } element;
};
```

Each `gxColor` structure holds the specification of a single color. Note that, besides the basic color-value formats such as `gxRGBColor` and `gxXYZColor`, a QuickDraw GX color can contain a 16-bit or 32-bit pixel value or an indexed color value, and you can also access the color as an array of color-component values. Each of the color values in the `element` union of the `gxColor` structure is described in the section “The Color Structure” beginning on page 4-53.

Color Conversion and Color Matching

Color support in QuickDraw GX is designed for device independence. You can work in whatever color space is most convenient for you, you can convert colors from one color space to another, and you can input and output colors with a variety of physical devices with minimum error and loss of information.

You may want to explicitly convert from one color space to another for a variety of reasons, such as

- n to allow users to work in a more familiar context (perhaps HSV instead of RGB)
- n to convert device-dependent colors to device-independent colors (such as RGB to L^*u^*v)
- n to preview printed output onscreen (by converting RGB to CMYK)
- n to display on monochrome monitors or printers (by converting to gray space)

In addition, QuickDraw GX automatically converts colors from one space to another whenever necessary, such as when it displays a color that is defined in terms of one space on a device whose colors are defined in terms of another space.

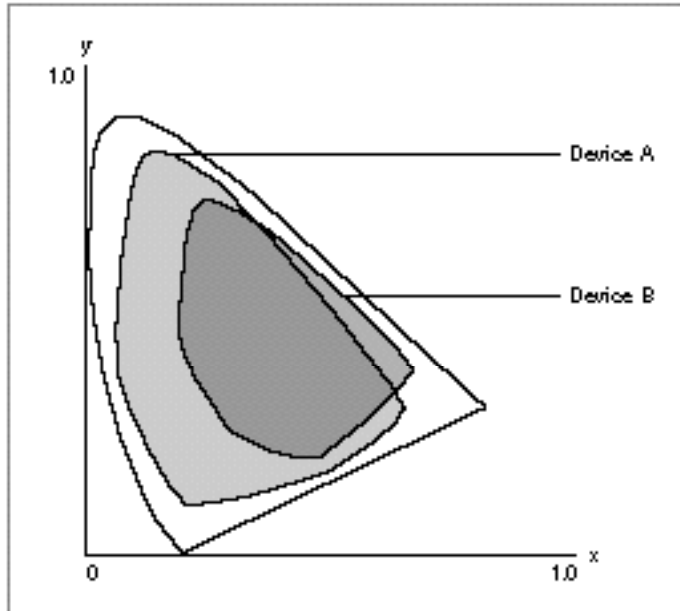
When converting among color spaces within a base family (such as HSV to RGB) and for display on the same device, the conversion is exact and there is no loss or error.

However, when converting across base families (such as RGB to CMYK, or HLS to XYZ), and when converting within the same family but across different display devices, device dependence is introduced and must be accounted for.

Different imaging devices (scanners, monitors, printers) work in different color spaces and each can have a different **gamut**, or range of colors that it can produce. Monitors from different manufacturers all display colors in RGB, but may have different RGB gamuts. Printers that work in CMYK space vary drastically in their gamuts, especially if they use different printing technologies. Even a single printer's gamut can vary significantly with the ink or type of paper it uses. It's easy to see that conversion from RGB colors on an individual monitor to CMYK colors on an individual printer using a particular paper type can lead to unpredictable results.

When an image is output to a particular device, the device displays only those colors that are within its gamut. Likewise, when an image is created by scanning, only those colors within the scanner's gamut are saved. Devices with different gamuts cannot reproduce each others' colors exactly, but careful shifting of the colors used on one device can improve the visual match when the image is displayed on another.

Figure 4-15 shows examples of two devices' color gamuts, projected onto Yxy space. Both devices produce less than the total possible range of colors, and device B is restricted to a significantly smaller range than device A. The problem illustrated by Figure 4-15 is to be able to display the same image on both devices with a minimum of visual mismatch. The solution to the problem is the use of color profiles and color-matching methods.

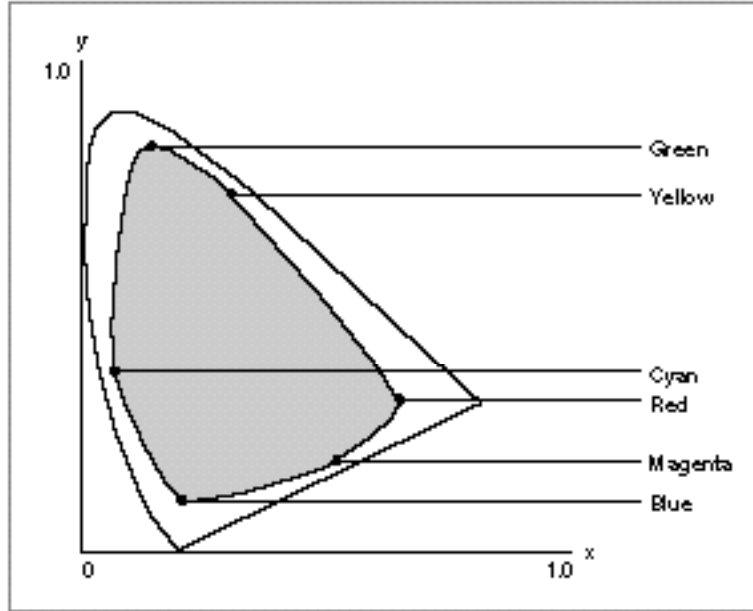
Figure 4-15 Color gamuts for two devices (in Yxy space)

Color Profiles

Converting colors accurately across different input or display devices is called **color matching**. To perform color matching requires the use of a color profile for each device involved. A **color profile** describes the characteristics of a color space for a particular physical device in a particular state. A monitor, for example, might have a single color profile, whereas a printer might have a different profile for each paper type or ink type it uses. A **color-matching method** uses a color profile to convert a color in a given color space on a given device to or from another color space or device, perhaps a device-independent color space.

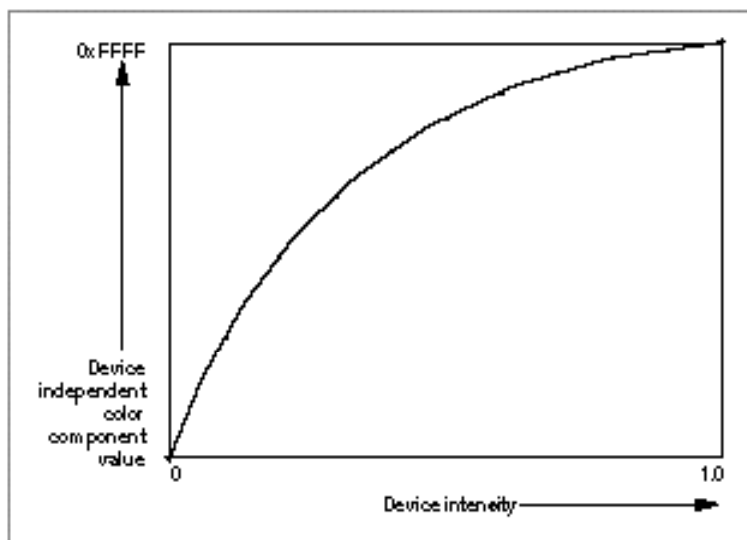
Different color profiles can have different kinds of information in them. However, any color profile has at least two parts: a set of profile chromaticities and a set of profile response curves. The **profile chromaticities** are color values that define the extremes of saturation that the device can produce for its primary and secondary colors (red, green, blue, cyan, magenta, yellow). Each color value is typically described in terms of a device-independent space such as XYZ. You can think of the profile's chromaticities as defining points at the extremes of that device's gamut, as shown in Figure 4-16. (The points in Figure 4-16 correspond to the limits of the gamut for device A in Figure 4-15.)

Figure 4-16 Profile chromaticities for a device (in Yxy space)



The **profile response curves** are graphs that describe how the profile chromaticities ramp from no intensity to full intensity (there are additional response curves for undercolor removal and black generation in CMYK space). The response curves are analogous to gamma curves for monitors or dot-pitch curves for printing. Figure 4-17 shows an example of a single response curve.

Figure 4-17 A profile response curve for a device



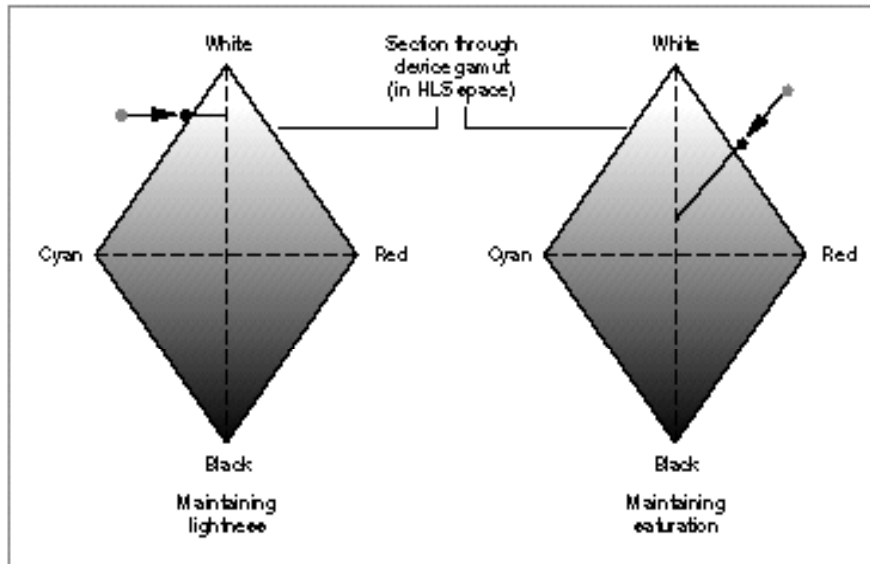
Color profiles contain additional information, such as a specification of how to apply the chromaticities and response curves for matching (see the next section, “Color-Matching Methods”), and a name string. They may also have custom information used by particular color-matching methods. QuickDraw GX uses color profiles following the format defined by the ColorSync Utilities. See the ColorSync Utilities chapter of *Inside Macintosh: Advanced Color Imaging* for more information.

Color profiles are optional; a given color structure may or may not contain a valid reference to a color profile. If a color profile reference is attached, QuickDraw GX uses it when converting or matching colors; if there is no attached profile, QuickDraw GX uses the default QuickDraw GX color profile; see “The Default Color Profile” on page 4-37.

Color-Matching Methods

When colors consistent with one device’s gamut are displayed on a device with a different gamut, as in Figure 4-15 on page 4-28, a color-matching method attempts to minimize the perceived differences in the displayed colors between the two devices. The default Apple color-matching method, as used with the ColorSync Utilities, uses these three approaches:

- n **Colorimetric matching.** In this method, colors that fall within the gamuts of both devices are left unchanged. For example, to match an image from device A onto device B in Figure 4-15, only the colors in the gamut of A that fall outside the gamut of B are altered. Colorimetric matching allows some colors in both images to be exactly the same, which is useful when colors must match quantitatively. A disadvantage of colorimetric matching is that many colors may map to a single color. All colors outside the gamut of B in Figure 4-15, for example, would be converted to colors at the edge of its gamut, reducing the total number of colors in the image and possibly greatly altering its appearance. In colorimetric matching, colors outside the gamut are usually converted to colors with the same lightness, but different saturation, at the edge of the gamut. The left side of Figure 4-18 shows how colors are projected in colorimetric matching.
- n **Perceptual matching.** In this method, all the colors of a given gamut are shifted to fit within another gamut. The colors maintain their relative positions, so the relationship between colors is maintained. With realistic images such as scanned photographs, perceptual matching produces better results than colorimetric matching in most cases; in Figure 4-15, for example, the eye could compensate for the difference in gamuts between A and B, and a perceptually matched image on B would look very similar to the original image on A. A disadvantage of perceptual matching is that none of the original colors is unchanged in the copy.
- n **Saturation matching.** In some computer graphics, such as bar graphs and pie charts, the actual color displayed is less important than its vividness. In this method, the relative saturation of colors is maintained from gamut to gamut. Colors outside the gamut are usually converted to colors with the same saturation, but different lightness, at the edge of the gamut. The right side of Figure 4-18 shows how colors are projected in saturation matching.

Figure 4-18 Maintaining lightness and maintaining saturation in color matching

QuickDraw GX uses the Macintosh ColorSync Utilities for color matching. ColorSync color-matching methods are Component Manager components and support all three kinds of color-matching, and may support other kinds as well. QuickDraw GX color profile objects contain ColorSync color profile structures, and each structure specifies the kind of matching that should be performed with it.

For more information on ColorSync and color-matching methods, see the ColorSync Utilities chapter of *Inside Macintosh: Advanced Color Imaging*. For more information on Component Manager components, see the Component Manager chapter of *Inside Macintosh: More Macintosh Toolbox*.

When Color Matching Occurs

Color profiles are associated with devices. For example, when a QuickDraw GX-aware scanning application creates a scanned image, it produces a bitmap and attaches a color profile object (containing profile information obtained from the scanner driver) to the bitmap. The color profile that is associated with a shape and describes the characteristics of the device on which the shape was created is called the **source profile**. If the colors in the bitmap are subsequently converted to another color space by the scanning application or by another QuickDraw GX application, QuickDraw GX uses that source profile to match the colors when converting. Bitmaps are described in the bitmap shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*.

To display the bitmap requires using another color profile, which is attached to the view device object associated with the output device. (View device objects are described in the chapter “View-Related Objects” in this book.) That color profile is called the **destination profile**. If the bitmap is displayed on a monitor, QuickDraw GX uses the monitor’s color profile, along with the bitmap’s source profile, to match the bitmap’s colors to the monitor’s gamut. If the bitmap is printed, QuickDraw GX uses the printer’s profile to match the bitmap’s colors to the printer, including generating black and removing undercolors where appropriate.

QuickDraw GX color matching occurs automatically, whenever drawing takes place or whenever colors are converted from a color space in one base family to a color space in a different base family. Most applications need not know what profiles, if any, are attached to the colors they manipulate and draw. However, applications can explicitly use color profiles for purposes such as print previewing, or they can allow the user to create custom, modified profiles for special purposes on particular devices. In addition, specialized applications can calibrate display devices and produce color profiles whose information is stored in the devices’ drivers for use by QuickDraw GX. Such applications make use of the ColorSync Utilities to create their profiles.

Color matching is off by default

Color matching can slow drawing speed. For that reason, when you create a view port, the view port attribute `gxEnableMatchPort` is cleared by default. If you want matching to occur when you draw to the screen, you must first set `gxEnableMatchPort`. (Matching occurs when appropriate during printing, regardless of the state of the `gxEnableMatchPort` attribute.) View port attributes are discussed in the chapter “View-Related Devices” in this book. \cup

For more information on color profiles, see the section “About Color Profile Objects” beginning on page 4-35.

About Color Set Objects

A color set is a QuickDraw GX object that contains a list of colors. Color sets exist to provide the colors for indexed color space. Bitmaps and other shapes that use indexed color space specify colors as indexes into a color set. Color sets are the QuickDraw GX equivalents to color tables in other graphics systems.

QuickDraw GX identifies an individual color set object through a color set **reference**. To obtain information about a color set object, you must send its reference as a parameter to a QuickDraw GX function (except that you can determine if two references identify the same color set object simply by comparing them for equality, and you can examine a reference to see if it is `nil`).

Any QuickDraw GX color (`gxColor` structure) that contains an indexed color value includes a reference to a color set object. If a shape's ink object has a color in indexed color space, the color includes a reference to a color set object. If the bitmap in a view device object uses indexed color values for its pixels, the bitmap includes a reference to a color set object. (View devices are described in the chapter "View-Related Objects" in this book.)

Color sets can be device independent because their colors, like any QuickDraw GX colors, can be matched across devices. The color information is valid for any display device on which the shapes the color sets apply to are drawn.

Color Set Properties

The interface to color set objects is entirely procedural. You manipulate the information in a color set by modifying its properties using QuickDraw GX functions.

Color set objects have four accessible properties, as shown in Figure 4-19. Note that, because a color set is an object and not a data structure, the order of the properties as shown in Figure 4-19 is completely arbitrary. Properties in italics are references to other objects.

Figure 4-19 The color set object and its properties



These are the four accessible properties in a color set:

- n **Color space.** The color space of all the color values in the color set. A color set can have only a single color space, which cannot be `gxIndexedSpace`.
- n **Color-value array.** An array of color values (not `gxColor` structures). Only the types of color values specified in the `gxSetColor` union are valid in a color set.
- n **Owner count.** The number of existing references to this color set object.
- n **Tag list.** A list of references to custom information about this color set object, stored in private data structures called *tag objects*. The chapter "Tag Objects" in this book describes tag objects in general and how you can use them to add custom information to objects.

QuickDraw GX provides functions to manipulate each of these properties. Note that there is no color profile property for a color set; profile information for the colors in a color set is found in the bitmap structure—or the color structure in the ink object—to which the color set is attached.

Color Values in a Color Set

The array of color values in a color set object can have up to 65,535 entries; each entry must be of one of the types defined in the `gxSetColor` union:

```
union gxSetColor{
    gxCMYKColor    cmyk;
    gxRGBColor     rgb;
    gxRGBAColor    rgba;
    gxHSVColor     hsv;
    gxHLSColor     hls;
    gxXYZColor     xyz;
    gxYXYColor     yxy;
    gxLUVColor     luv;
    gxLABColor     lab;
    gxYIQColor     yiq;
    gxColorValue   gray;
    gxGrayAColor   graya;
    unsigned short pixel16;
    unsigned long  pixel32;
    gxColorValue   component[4];
};
```

The `gxSetColor` union is an abbreviated color structure (see page 4-53). It has no `profile` or `space` fields, because individual colors within a color set cannot have different color spaces, and because the color profiles for the color values are defined elsewhere—in the individual colors, bitmaps, or transfer modes that use this color set. Also, the `gxSetColor` union has no `gxIndexedColor` field because color sets cannot be recursive (that is, colors in a color set cannot refer to colors in other color sets).

Default Color Sets

QuickDraw GX maintains several default color sets, one for each possible pixel size in bitmaps that use indexed space—1, 2, 4, and 8 bits. (Bitmaps with pixel sizes over 8 bits cannot use indexed space.) When you create a bitmap with a pixel size of 8 bits or less and specify `nil` for its color set, QuickDraw GX uses the appropriate default color set whenever you draw that bitmap.

Each of the default color sets consists of a gray ramp, using color values in `gxGraySpace` that progress in order from white at an index value of 1 to black at the highest index value. For a pixel size of 1 bit, for example, the default color set consists of two colors: white and black.

You do not create a copy of any of the default color sets by calling the `GXNewColorSet` function; that function requires you to supply a specific array of color values.

You can inspect and change any of the default color sets by using the `GXGetDefaultColorSet` function, described on page 4-62, and the `GXSetDefaultColorSet` function, described on page 4-63. Bitmaps are described in the bitmap shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*.

About Color Profile Objects

A color profile is a QuickDraw GX object that describes the color response of a specific device, class of device, or device configuration. As described in the section “Color Profiles” beginning on page 4-28, a color profile provides a quantitative description of a device’s color gamut in terms of standard, usually device-independent colors. QuickDraw GX uses color profiles for color matching when converting colors and when drawing shapes.

QuickDraw GX identifies a color profile object through a color profile **reference**. To obtain information about a color profile, you must send its reference as a parameter to a QuickDraw GX function (except that you can determine if two references identify the same color profile object simply by comparing them for equality, and you can examine a reference to see if it is `nil`).

Any QuickDraw GX color (`gxColor` structure), such as the color in a shape’s ink object, can include a reference to a color profile object. Any bitmap structure, including the bitmap in a view device object, can reference a color profile. (View devices are described in the chapter “View-Related Objects” in this book.) A transfer mode structure can also reference a color profile. If a color, bitmap, or transfer mode contains no specific reference to a color profile, QuickDraw GX uses a default profile when converting colors and when drawing.

Even though color profiles are inherently device-specific, QuickDraw GX uses them consistently and performs color matching when needed. Most applications need not pay attention to color profiles or try to associate them with specific devices except when first creating colors. If you create a color, you should attach to it a color profile that describes the characteristics of the device on which the color was created. If the device’s characteristics are equivalent to the Apple 13-inch color monitor—or if you never need to display or print the color on another device—you need not attach a profile.

Color Profile Properties

The interface to color profile objects is entirely procedural. You manipulate the information in a color profile by modifying its properties using QuickDraw GX functions.

Color profile objects have three accessible properties, as shown in Figure 4-20. Note that, because a color profile is an object and not a data structure, the order of the properties as shown in Figure 4-20 is completely arbitrary. Properties in italics are references to other objects.

Figure 4-20 The color profile object and its properties



These are the three accessible properties in a color profile object:

- n **Profile data.** Information specific to the individual profile, that usually includes color values and a set of data that plots the response of the device—from zero intensity to full intensity—when it generates each of the specified colors.
- n **Owner count.** The number of existing references to this color profile object.
- n **Tag list.** A list of references to custom information about this color profile object, stored in private data structures called *tag objects*. The chapter “Tag Objects” in this book describes tag objects in general and how you can use them to add custom information to objects.

QuickDraw GX provides functions to manipulate each of these properties.

Profile Data

The profile data is the actual color profile information, in the form of a ColorSync color profile structure. A QuickDraw GX color profile object is a wrapper for a ColorSync profile.

ColorSync profiles are specified by the `CMProfile` structure, which consists of the following parts:

- n **Header.** A structure containing information such as the size, version, device type, and attributes of the profile. The header also contains the XYZ chromaticities of the device’s white point and black point, and an options field that specifies the type of color matching preferred (such as perceptual, colorimetric, or saturation matching).

Colors and Color-Related Objects

- n **Profile chromaticities.** A structure that contains the XYZ chromaticities for the six primary and secondary colors (red, green, blue, cyan, magenta, yellow) at the limits of the device's gamut.
- n **Profile response curves.** A variable-sized array of response curves for each of the primary and secondary colors, plus gray (plus black generation and undercolor removal for printer profiles).
- n **Name string.** An international string, which consists of a Macintosh script code followed by a 63-byte text string, that identifies the profile. (Note that these are Macintosh script codes, which differ from QuickDraw GX script codes; Macintosh script codes are described in the Script Manager chapter of *Inside Macintosh: Text*.)
- n **Custom data.** Information used by custom color-matching methods. It may include other kinds of color values or response curves.

The details of the `CMProfile` structure, including explanations of some of the terms used here, are given in the ColorSync Utilities chapter of *Inside Macintosh: Advanced Color Imaging*. All parts of the structure except for the custom data are accessible through ColorSync function calls. QuickDraw GX defines no structures or types for the profile data of a color profile object, although you can access the information if you know its format. See “Manipulating the Profile Data in a Color Profile Object” beginning on page 4-48.

The Default Color Profile

QuickDraw GX maintains a default color profile that it uses for color matching when no color profile is explicitly provided—that is, when the `profile` field of a color structure or bitmap structure is `nil`. The default color profile reflects the color response of the Apple 13-inch color monitor, as defined by ColorSync version 1.0.

You do not create a copy of the default color profile by calling the `GXNewColorProfile` function; that function requires you to supply profile data for the object you are creating. Also, you cannot change the characteristics of the default color profile object; there is no `GXSetDefaultColorProfile` function.

You can determine the actual profile chromaticities used for the default QuickDraw GX color profile by retrieving it and examining its profile data.

Zero-Length Profiles

QuickDraw GX automatically performs color matching whenever it draws or converts colors, and if you specify a `nil` color profile reference in any situation, QuickDraw GX uses the default color profile rather than using no profile.

In some cases, however, you may want to prevent color matching from occurring for individual colors or shapes, such as when comparing or calibrating different devices. To do so, you can use a zero-length profile. A **zero-length profile** is a color profile object in which the profile data is of zero length. It is a valid QuickDraw GX object—its reference is not `nil`—but it contains no data. If you attach a zero-length profile to a color, QuickDraw GX performs no matching when that color is drawn or converted.

If, for example, you want to see how each attached device represents pure blue, you can specify pure blue for an ink's color, attach a zero-length profile to it, and draw a shape with that color to each device. If instead you specify `nil` for the profile when creating a color, QuickDraw GX matches the color, using the default color profile and each device's color profile, when drawing.

In the case of color conversions that require a profile (those between base families, such as from RGB to XYZ), QuickDraw GX uses the following conventions:

- n If both profiles are zero-length, QuickDraw GX uses the default profile as both the source and the destination profile.
- n If only one profile is zero-length, QuickDraw GX uses the other profile as both the source and the destination profile.

Note

To turn off color matching entirely when drawing to a view port, make sure that the `gxEnableMatchPort` attribute for that view port is cleared. (It is cleared by default.) View port attributes are discussed in the chapter "View-Related Devices" in this book. [u](#)

You can create a zero-length profile using the `GXNewColorProfile` function, described on page 4-79, or the `GXSetColorProfile` function, described on page 4-89.

Using Colors and Color-Related Objects

This section describes how to create and use colors, color sets, and color profile. It shows how you can

- n assign colors to shapes and color profiles to colors
- n test and compare colors
- n convert colors from one color space to another, and apply color matching when converting and when scanning, displaying, or printing
- n create and manipulate color set objects, to support indexed colors
- n create and manipulate color profile objects, to support color matching

Assigning Colors to Shapes

Colors exist to affect the appearance of drawn shapes. QuickDraw GX shapes other than bitmaps and pictures get their color from the ink object that is part of the shape. One property of the ink object is `color`, a `gxColor` structure that describes the color of the associated shape.

To assign or change a shape's color, therefore, you typically call the `GXSetInkColor` function for the ink associated with the shape whose color you are assigning. You can also call `GXSetShapeColor`, which performs the same task but allows you the convenience of specifying the shape object involved, rather than the ink object that actually contains the color information. (Conversely, to inspect the color of a shape, you call `GXGetInkColor` or `GXGetShapeColor`.) The `GXSetInkColor`, `GXSetShapeColor`, `GXGetInkColor`, and `GXGetShapeColor` functions are described in the chapter "Ink Objects" in this book.

Shapes that need more than one color are a special case. Bitmap shapes do not use the color information in their ink object. Instead, the color of each pixel in a bitmap shape is specified as a pixel value in the `gxBitmap` structure; depending on the storage size of each pixel, that pixel value may be an actual color value or it may be an index into a color set. To set the color of an individual pixel in a bitmap, you call the `GXSetShapePixel` function, specifying which pixel to modify and what its new color or new index value is. (Conversely, you can inspect the color of a pixel by calling `GXGetShapePixel`.)

Modifying the color values in a color set, as described in the section "Manipulating the Colors in a Color Set Object" on page 4-47, is another way to change the color or colors of a shape. In a bitmap using indexed color space, any pixels whose indexes refer to color values you have modified will be changed in appearance, even though their pixel values remain unchanged. You can use this technique to perform simple manipulations of a shape's colors.

Bitmap shapes, the `gxBitmap` structure, and the functions `GXSetShapePixel` and `GXGetShapePixel` are described in the bitmap shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*.

Assigning Color Profiles to Colors

When the user creates or modifies shapes' colors, you assign colors to the ink objects or pixels associated with those shapes. To assure proper color matching, you can assign a color profile to each color or bitmap that the user creates. Normally, the user works with a monitor attached to the system; you can find the profile for that monitor by examining the `bitmap` property of the view device associated with the view port the user draws into. You can attach that profile to the user's colors. In the case of a single shape displayed on more than one device, you may have to pick (or allow the user to pick) which view device is the controlling one. In a `gxColor` structure or a `gxBitmap` structure, you place the color profile reference in the `profile` field.

If you assign no color profile to a color or bitmap, Quickdraw GX uses the default color profile when drawing or converting.

If you want to make sure that no matching occurs, assign a zero-length profile to the color or bitmap. A zero-length profile is a color profile object whose profile data is of zero length.

Comparing and Testing Colors

QuickDraw GX provides several functions that allow you to analyze individual color values for various purposes.

Checking for Out-of-Gamut Colors

If you have a color value that you want to test against a given color space or color set, you can use the `GXCheckColor` function. For example, you can use `GXCheckColor` to see if a given color is representable on a particular printer. If the color is not directly representable—that is, if it is **out of gamut**—you could alert the user to that fact. You could also call the `GXConvertColor` function to mimic the automatic color conversion that would take place in printing, to determine what color the printer would use to represent your given color.

Both `GXCheckColor` and `GXConvertColor` require the color space and color profile of the device the color is destined for. To get the color space and color profile of a printer, you can use the `GXGetPrinterViewDevice` and `GXGetViewDeviceBitmap` functions.

The `GXCheckColor` function is described on page 4-57. The `GXConvertColor` function is described on page 4-60. The `GXGetPrinterViewDevice` function is described in *Inside Macintosh: QuickDraw GX Printing*. The `GXGetViewDeviceBitmap` function is described in the chapter “View-Related Objects” in this book.

Checking Colors for Closeness and Color Space

If you want to compare a user-selected color with the range of colors in a color set, you can use the `GXGetColorDistance` function to determine how far the selected color is from any of the colors in the color set. If the selected color is close enough (in color-space distance) to one of the existing colors in the color set, you could call the `GXConvertColor` function to change the selected color to that closest color. Alternatively, you could call `GXGetColorSetParts` and `GXSetColorSetParts` to add the selected color to the color set or replace another color in the color set with the selected color.

As another example, suppose that you open a document containing shapes of various colors, and you want to save a grayscale version of that document. You might call `GXCheckColor` on each color in the document, and then `GXConvertColor` on each color whose color space is not already `gxGraySpace`. (You might also save the original color information as a tag object attached to each shape or ink, for later restoration.)

The `GXGetColorDistance` function is described on page 4-58. The `GXConvertColor` function is described on page 4-60. The `GXGetColorSetParts` function is described on page 4-75; the `GXSetColorSetParts` function is described on page 4-76. The `GXCheckColor` function is described on page 4-57.

Predicting Drawing Results

You can preflight, or predict, the results of a drawing operation by using the `GXCombineColor` function. You supply a destination color, and `GXCombineColor` tells you what would happen if a shape using the ink object you specify were drawn to a destination of that color. This function is as much a test of transfer mode as it is of source and destination colors; you can use it to see how, or even if, drawing occurs under the conditions you specify. For example, if you are using the `gxMigrateMode` transfer mode, you may want to adjust the operand so that the result color exactly equals the source color for a particular destination color. You can call `GXCombineColor` with different operand values until you get the result you want, and then draw the actual shape. Transfer modes, operands, and source and destination colors are described in the chapter “Ink Objects” in this book.

The `GXCombineColor` function is described on page 4-59.

Converting and Matching Colors

Although conversion among color spaces happens automatically whenever necessary during the drawing process, you can also explicitly convert colors if you need to. The following code fragment uses `GXConvertColor` to modify the hue of a shape, preserving its luminance and saturation. Such a technique is one way to perform color animation. The code gets the color of shape `theShape`, converts it to HSV space, increases the hue value just enough to make a perceptible difference, reassigns the color to the shape, and draws the shape:

```
gxColor    oldColor;
GXGetShapeColor(theShape, &oldColor);
GXConvertColor(&oldColor, hsvSpace, nil, nil);
oldColor.element.hsv.hue += 0x300;
GXSetShapeColor(theShape, &oldColor);
GXDrawShape(theShape);
```

(Note that the final `nil` parameter in the call to `GXConvertColor` means that the converted color reassigned to the shape uses the default color profile, whether or not the original one did.)

Color matching happens automatically whenever you draw a shape or convert colors. If the profile reference in a color is `nil`, color correction still occurs when needed, as when converting from RGB to CMYK color space. In those cases, the default profile is used.

In some cases, you may want to prevent color matching from occurring for an individual color, such as when comparing or calibrating different devices. If you attach a zero-length profile to a color, QuickDraw GX performs no matching when that color is drawn or converted to another color space.

To prevent color matching from occurring during all drawing to a given view port, clear the `gxEnableMatchPort` attribute of that view port. Note that, because color matching can slow down the drawing process, this attribute is cleared by default on all view ports.

Therefore, if you want color matching to occur when drawing to the screen, you must explicitly set `gxEnableMatchPort`. Even if you do want matching to occur, you might still clear `gxEnableMatchPort` temporarily during scrolling or other repetitive drawing processes. (For printing, QuickDraw GX automatically takes care of making sure that color matching occurs when it is needed.)

If you want to specify a particular kind of color-matching method other than the one specified in the profile attached to the color you are matching, your application can either modify the information in the color profile object using QuickDraw GX calls, or make calls to the ColorSync Utilities to specify the one you want.

To allow the user to preview on the screen what printing would look like, you can mimic on the monitor the profile characteristics of the printer. You need to convert the color you are drawing to the color space of the printer—applying the printer’s color profile—and then convert that color back to the monitor’s color space—applying the monitor’s color profile—and then draw. One way to do that is to create an offscreen view group with the printer’s color space and color profile, and draw into a view port in that view group. Then, draw from the bitmap of the offscreen view port into the view port of the monitor.

Color matching is discussed in the section “Color Conversion and Color Matching” beginning on page 4-26. Color profiles, the default profile, and zero-length profiles are discussed in the section “About Color Profile Objects” beginning on page 4-35. The `gxEnableMatchPort` view port attribute is described in the chapter “View-Related Devices” in this book.

Creating and Manipulating Color Set and Color Profile Objects

This section describes how you can create and interact with color set objects and color profile objects as whole entities—to create, dispose of, copy, compare, clone, load, and unload them. Because color sets and color profiles are QuickDraw GX objects, and you use similar sets of functions to manipulate them, they are considered together in each of the subsequent sections. Manipulating the individual properties of color sets and color profiles is described under “Manipulating Object Properties of Color Sets and Color Profiles” beginning on page 4-46.

Creating and Disposing of a Color Set or Color Profile

QuickDraw GX provides the `GXNewColorSet` function to allow you to create a new color set. You can also create a new color set that is a copy of an existing color set by calling `GXCopyToColorSet`.

Once you have created a color set object, you can attach it to a color structure, bitmap structure, or transfer mode structure by putting a reference to it in the color or bitmap or transfer mode. Colors, bitmaps, and transfer modes also include a specification of the color space they use; if they use a color set, they must use `gxIndexedSpace` for their color space.

Colors and Color-Related Objects

The following code fragment creates a simple color set (`theSet`) with two RGB colors: black and white. It assigns the color set to the bitmap structure `theBits`, which it then assigns to the bitmap shape `theBitmap`, which it finally assigns to the view device `theDevice`. The code then disposes of the color set and bitmap shape since those references are no longer needed:

```

gxSetColor  theColors[2];
gxSetColor  *pColor;
gxColorSet  theSet;
gxBitmap    theBits
gxShape     theBitmap
.
.  /* initialize theBits and theBitmap (not shown) */
.
pColor = &theColors[0];
pColor->rgb.red = pColor->rgb.green =
                pColor->rgb.blue = gxColorValue1;
pColor++;
pColor->rgb.red = pColor->rgb.green = pColor->rgb.blue = 0x0000;

theSet = GXNewColorSet(gxRGBSpace, 2, theColors);
theBits.set = theSet;
GXSetBitmap(theBitmap, &theBits, nil);
GXSetViewDeviceBitmap(theDevice, theBitmap);
GXDisposeColorSet(theSet);
GXDisposeShape(theBitmap);

```

Note

If you use `GXNewColorSet` to create a color set, and then assign it as a default color set with `GXSetDefaultColorSet`, be sure to dispose of your reference to that color set immediately after assigning it as the default. That way the new default color set will have the proper owner count of 1, as required by QuickDraw GX. ^u

For color profile objects, QuickDraw GX provides the `GXNewColorProfile` function (and the `GXCopyToColorProfile` function) to allow you to create new color profiles. If you have profile information that you want to attach to a color or to a bitmap, you can put that information in object form with `GXNewColorProfile` and attach it (by reference) to the color or bitmap. For simple drawing, you typically never have to do this, but you might want to create a color profile object in these special instances:

- n If you want to inhibit color matching for a particular color, you can create a zero-length profile (one with no profile data) and attach it to the color.
- n If you have access to a profile structure, either as a resource or through calls to the ColorSync Utilities, you can turn that structure into a QuickDraw GX color profile by creating a color profile object with that structure as the profile data.

Colors and Color-Related Objects

- n If your application is a scanning application, it can create a color profile object from information in the scanner's driver and attach that profile to the bitmap shapes it creates.
- n If your application is a calibration program that develops profile information for a device, it can create a color profile object to hold the profile information it generates during the calibration process and to display the results to the operator of the calibration program.

If your program is a device driver, it contains profile information in the form of color profile resources; it does not need to create color profile objects. How device drivers store color profile information is described in the printing resources chapter of *Inside Macintosh: QuickDraw GX Printing Extensions and Printer Drivers*.

To delete your application's reference to a color set or color profile object, call the `GXDisposeColorSet` or `GXDisposeColorProfile` function. Calling either function may or may not actually release the memory allocated for the object, depending on the object's owner count. Both of these functions decrease the owner count of the color set or color profile by 1; if that brings the owner count to zero, the object is completely deleted and its memory released. See "Manipulating Owner Counts" on page 4-46.

The `GXNewColorSet` function is described on page 4-64; the `GXNewColorProfile` function is described on page 4-79. The `GXDisposeColorSet` function is described on page 4-65; the `GXDisposeColorProfile` function is described on page 4-80.

Copying, Comparing, and Cloning Color Sets and Color Profiles

You can use the `GXCopyToColorSet` function to copy color information from one color set object to another or to create a new copy of an existing color set. You can use the `GXCopyToColorProfile` function to copy profile information from one color profile object to another or to create a new copy of an existing color profile.

You can test if two references refer to the same color set or color profile object by simply comparing the references for equality. You can also test two different color set or color profile objects for equality with the `GXEqualColorSet` and `GXEqualColorProfile` functions, respectively. For two color sets to be equal, their color spaces and colors must be identical; for two color profiles to be equal, their profile information and their attributes must be equal. In either case, the common object properties (owner count and tag list) do not need to be identical for the objects to be considered equal.

Object copies created with the `GXCopyToColorSet` and `GXCopyToColorProfile` functions are always equal, in terms of the criteria just listed, to the objects from which they were copied.

In certain circumstances, you may want to copy a reference to a color set or color profile without actually copying the object. For example, you may want two variables to refer to the same color set or color profile object, so that altering one of them affects both. This is called **cloning** an object, rather than copying it. You can use the `GXCloneColorSet` and `GXCloneColorProfile` functions to clone a color set or color profile, respectively.

Colors and Color-Related Objects

Functionally, `GXCloneColorSet` and `GXCloneColorProfile` do nothing more than increase the owner count of the specified object. For more information about cloning objects, see the chapter “Introduction to Objects” in this book. For information on manipulating owner counts, see the section “Manipulating Owner Counts” on page 4-46.

The following code fragment initializes a bitmap structure to be used for offscreen drawing, assigns a color set object (`commonColorSet`) to it, and then creates a bitmap shape (`shMap`) with that bitmap. The code, for its own purposes of tracking owner count (not shown here), clones the color set rather than just assigning it to the bitmap shape. In general, cloning is not necessary when you assign a color set to a bitmap, because when you then call `GXNewBitmap` to create the bitmap shape (as this code fragment does), `QuickDraw GX` increases the color set’s owner count for you.

```
gxBitmap          map;
gxPoint           pt = {0, 0};
gxShape           shMap = nil;
.
. /* set the bitmap's width, height, and pixel size */
.
map.rowBytes = 0L;
map.image = nil;
map.space = gxIndexedSpace;
map.profile = nil;
map.set = GXCloneColorSet(commonColorSet);
shMap = GXNewBitmap(&map, &pt);
```

`QuickDraw GX` will decrease the owner count of the color set when the shape `shMap` is disposed of, but the application code will also need to call `GXDisposeColorSet` at some point, to balance the `GXCloneColorSet` call it makes here.

The `GXCopyToColorSet` function is described on page 4-66; the `GXCopyToColorProfile` function is described on page 4-81. The `GXEqualColorSet` function is described on page 4-67; the `GXEqualColorProfile` function is described on page 4-82. The `GXCloneColorSet` function is described on page 4-68; the `GXCloneColorProfile` function is described on page 4-83.

Loading and Unloading Color Sets and Color Profiles

Although you rarely need to, you can influence memory-allocation decisions involving objects that you have created. If your application needs to have a color set object or color profile object in memory, it can force `QuickDraw GX` to load the object into memory. When your application no longer needs the color set or color profile in a loaded state, it can instruct `QuickDraw GX` to unload the object.

Colors and Color-Related Objects

You call the `GXLoadColorSet` or `GXLoadColorProfile` function to make sure that a color set or color profile object is in memory; if it has been unloaded, QuickDraw GX brings it into memory. You can call the `GXUnloadColorSet` or `GXUnloadColorProfile` function to instruct QuickDraw GX that it is free to unload the color set or color profile at any time. These functions are described in the memory management chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

Manipulating Object Properties of Color Sets and Color Profiles

This section describes how to manipulate the common object properties of color sets and color profiles: owner count and tag list. It also describes how to manipulate the colors of a color set and the profile data of a color profile.

For manipulating color sets and color profiles as whole objects, see “Creating and Manipulating Color Set and Color Profile Objects” beginning on page 4-42.

Manipulating Owner Counts

The owner count of an object indicates the number of current references to that object. In general, QuickDraw GX manages owner counts for you. For example, when you create a new color set object, QuickDraw GX sets the owner count of the new color set to 1. If you assign that color profile object to a bitmap structure and then assign that bitmap structure to a bitmap shape, QuickDraw GX increments the color profile’s owner count, corresponding to the new reference to the color profile contained in the bitmap structure.

In some situations, as when switching color profiles or color sets among objects that reference them, you may want to directly manage their owner counts yourself. To do so, you can

- n use the functions `GXGetColorSetOwners` or `GXGetColorProfileOwners` to determine the current owner count
- n use the functions `GXCloneColorSet` or `GXCloneColorProfile` to increment the owner count whenever you create a new reference to the object
- n use the functions `GXDisposeColorSet` or `GXDisposeColorProfile` to decrement the owner count, freeing the memory used by the color set or color profile if the owner count goes to 0

The code fragment on page 4-45 shows an example of an application explicitly managing the owner count of a color profile object.

The `GXGetColorSetOwners` function is described on page 4-69. The `GXGetColorProfileOwners` function is described on page 4-84.

In the chapter “Style Objects” in this book, the section on manipulating a style object’s owner count discusses two common owner-count problems and how to avoid them. The problems are discussed in terms of style objects, but they apply equally well to color sets and color profiles. Refer to that discussion if you find that the color-related objects you create have owner counts that are higher or lower than you expect.

Getting and Setting Tag References

You can examine the list of references to tag objects currently associated with a color set object or color profile object by using the `GXGetColorSetTags` or `GXGetColorProfileTags` function. Once you create a tag object, you can attach it to its object using the `GXSetColorSetTags` or `GXSetColorProfileTags` function. You can attach as many tag objects as you like to a color set or color profile.

Tag objects and the basic functions for manipulating them are described in the chapter “Tag Objects” in this book. That chapter also lists the common tag types defined and reserved by Apple Computer, Inc.

The `GXGetColorSetTags` function is described on page 4-70; the `GXGetColorProfileTags` function is described on page 4-85.

The `GXSetColorSetTags` function is described on page 4-71; the `GXSetColorProfileTags` function is described on page 4-86.

Manipulating the Colors in a Color Set Object

If you are using indexed color space, you can gain access to the array of colors in the space’s color set or to any contiguous subset of the colors in the array. You can then inspect, rearrange, modify, or add or delete colors from the array.

For example, suppose you want to sort the colors in a color set so that they will display in a visually useful manner in a palette for the user. You could first call the `GXGetColorSet` function to get the array of colors. You could then sort the colors (say, by hue (H) in `gxHSVSpace`), and then return the array to the color set by calling the `GXSetColorSet` function.

Alternatively, suppose you already have a luminance-sorted array of colors in a color set, and you want to convert the first (darkest) color in the array to pure black. Instead of accessing the entire array, you can call `GXGetColorSetParts` to get only the first color in the array. You can then change that color to black, and reinsert it in the color set by calling `GXSetColorSetParts`.

To add colors to or delete colors from a color set, call `GXGetColorSet`, modify the color-value array as needed, and then call `GXSetColorSet` to place the new array in the color set.

To change the color space of a color set, follow this sequence of calls:

- n Call `GXGetColorSet` to obtain the color-value array.
- n Call `GXConvertColor` on each color value in the array to convert the individual color values from one space to the other.
- n Call `GXSetColorSet` to place the same array in the color set, but with a different value specified for the color space.

Remember that simply changing the color space of a color set does not convert the individual color values from one space to the other.

As an example of color-set manipulation, the following code fragment from a drawing routine matches each of the colors of a color set used by the shape `matchShape` to a specific color profile (`qmsProfile`). The code uses the `GXGetColorSet` function to fill out a temporary array of color values (`mycolors`) from the color set, converts each color (from RGB space with a `nil` profile to RGB space with `qmsProfile`, in this case) with the `GXConvertColor` function, and then reassigns the color values to the color set with the `GXSetColorSet` function.

```
gxSetColor mycolors[256];
oldColorCount = GXGetColorSet(GXGetShapeColorSet(matchShape),
                              nil, mycolors);
for (i = 0; i < oldColorCount; i++)
{
    gxColor tmpColor;
    tmpColor.space = gxRGBSpace;
    tmpColor.profile = nil;
    tmpColor.element.rgb = mycolors[i].rgb;
    GXConvertColor(&tmpColor, gxRGBSpace, nil, qmsProfile);
    mycolors[i].rgb = tmpColor.element.rgb;
}
GXSetColorSet(GXGetShapeColorSet(matchShape), gxRGBSpace,
              oldColorCount, mycolors);
```

The `GXGetColorSet` function is described on page 4-73. The `GXSetColorSet` function is described on page 4-74. The `GXGetColorSetParts` function is described on page 4-75. The `GXSetColorSetParts` function is described on page 4-76.

Manipulating the Profile Data in a Color Profile Object

QuickDraw GX defines no structures or types for the profile data of a color profile object. For drawing or converting colors, most applications have no need to access or alter the data in a color profile object. For special needs, however, such as changing the type of match you want to perform, using a custom color-matching method, or inspecting the name of a profile, you can—with knowledge of the details of the `CMProfile` structure—access and alter the profile data of an existing color profile object. Also, if your application is a calibration program that creates color profiles for devices, or if it is an imaging application that allows users to customize color profiles for specific uses, you need access to profile information in order to make or modify a color profile object.

One way to do this is to use `ColorSync` functions to manipulate a `ColorSync` profile directly, and then use the QuickDraw GX function `GXNewColorProfile` to convert it to a color profile object. `ColorSync` profiles are commonly in the `ColorSync profiles` folder on the user's system, and `ColorSync` can provide you with a list of those profiles.

More directly, you can call the `GXGetColorProfile` function to obtain the profile data for a given profile. Knowing the structure of a `ColorSync` color profile, you can then modify that information as needed, and return the altered data to the color profile object by calling the `GXSetColorProfile` function.

Note

If you alter the header of a `ColorSync` color profile to specify a particular color space in the `dataType` field, and then apply that profile to a color defined in terms of a different color space, QuickDraw GX ignores the new header data and specifies the color space implied by the color value you pass to the profile. ^u

Yet another approach is to directly modify the profile data of a color profile object in place, in QuickDraw GX memory. First, you call the `GXLockColorProfile` function to prevent the profile data from being relocated, and then you call `GXGetColorProfileStructure` to get a pointer to the profile data. After manipulating the data, you must call `GXUnlockColorProfile` to release the data for relocation. Remember that you cannot change the *size* of the profile data with these calls, only its contents; if your manipulations require a change in the size of the data, you must use `GXGetColorProfile` and `GXSetColorProfile`.

IMPORTANT

Memory-handling complications can occur with locked objects. Locking an object fragments the QuickDraw GX heap, which can result in lower performance. Furthermore, if a fragmented-memory condition occurs during a call, QuickDraw GX may unlock all objects and restart the call. Therefore, be careful about performing memory-intensive operations while there are locked objects in QuickDraw GX memory; they may become unlocked without warning. ^s

The `GXNewColorProfile` function is described on page 4-79. The `GXGetColorProfile` function is described on page 4-88. The `GXSetColorProfile` function is described on page 4-89. The `GXLockColorProfile` function is described on page 4-90. The `GXGetColorProfileStructure` function is described on page 4-92. The `GXUnlockColorProfile` function is described on page 4-91.

Colors and Color-Related Objects Reference

This section provides reference information to the data structures and functions that allow you to work with colors and create and manipulate color sets and color profiles, and to alter their properties. It describes

- n the constants and data types that define colors and color-related objects
- n the QuickDraw GX functions that operate on colors
- n the QuickDraw GX functions that operate on color sets
- n the QuickDraw GX functions that operate on color profiles

Constants and Data Types

This section describes the constants and data types that define

- n colors and color spaces
- n color set objects
- n color profile objects

Color-Component Values

Each color component in a color space (other than an indexed color space) is described by a numeric color-component value, defined by the `gxColorValue` type definition:

```
typedef unsigned short gxColorValue;
```

Color-component values can vary from 0 (no intensity) to 0xFFFF (maximum intensity). You can use the constant `gxColorValue1` to represent 0xFFFF.

Color Values

Color-component values combine to form color values. Each color value is a complete specification of a single color in a given color space. QuickDraw GX recognizes the following ten fundamental types of color values:

- n **CMYK color value.** It contains color-component values for cyan, magenta, yellow, and black. It is defined by the `gxCMYKColor` type definition:

```
struct gxCMYKColor{
    gxColorValue    cyan;
    gxColorValue    magenta;
    gxColorValue    yellow;
    gxColorValue    black;
};
```

- n **RGB color value.** It contains color-component values for red, green, and blue. It is defined by the `gxRGBColor` type definition:

```
struct gxRGBColor{
    gxColorValue    red;
    gxColorValue    green;
    gxColorValue    blue;
};
```


Colors and Color-Related Objects

- n **Alpha-channel RGB color value.** It contains color-component values for red, green, and blue, plus a fourth (alpha) color-component value representing opacity. It is defined by the `gxRGBAColor` type definition:

```
struct gxRGBAColor{
    gxColorValue    red;
    gxColorValue    green;
    gxColorValue    blue;
    gxColorValue    alpha;
};
```

- n **HSV color value.** It contains color-component values for hue, saturation, and value. It is defined by the `gxHSVColor` type definition:

```
struct gxHSVColor{
    gxColorValue    hue;
    gxColorValue    saturation;
    gxColorValue    value;
};
```

- n **HLS color value.** It contains color-component values for hue, lightness, and saturation. It is defined by the `gxHLSColor` type definition:

```
struct gxHLSColor{
    gxColorValue    hue;
    gxColorValue    lightness;
    gxColorValue    saturation;
};
```

- n **XYZ color value.** It contains color-component values for the X, Y, and Z tristimulus values. It is defined by the `gxXYZColor` type definition:

```
struct gxXYZColor {
    gxColorValue    x;
    gxColorValue    y;
    gxColorValue    z;
};
```

- n **Yxy color value.** It contains color-component values for the Y, x, and y chromaticity axes. (Note that the Y component is identified in this color structure as `capY`.) It is defined by the `gxYXYColor` type definition:

```
struct gxYXYColor {
    gxColorValue    capY;
    gxColorValue    x;
    gxColorValue    y;
};
```

Colors and Color-Related Objects

- n **L*u*v* color value.** It contains color-component values for the L*, u*, and v* axes. It is defined by the `gxLUVColor` type definition:

```
struct gxLUVColor {
    gxColorValue  l;
    gxColorValue  u;
    gxColorValue  v;
};
```

- n **L*a*b* color value.** It contains color-component values for the L*, a*, and b* axes. It is defined by the `gxLABColor` type definition:

```
struct gxLABColor {
    gxColorValue  l;
    gxColorValue  a;
    gxColorValue  b;
};
```

- n **YIQ color value.** It contains color-component values for the Y, I, and Q axes. It is defined by the `gxYIQColor` type definition:

```
struct gxYIQColor{
    gxColorValue  y;
    gxColorValue  i;
    gxColorValue  q;
};
```

- n **Grayscale color value.** It contains a single color-component value for luminance.
- n **Alpha-channel grayscale color value,** containing a color-component value for luminance, plus a second (alpha) color-component value representing opacity. It is defined by the `gxGrayAColor` type definition:

```
struct gxGrayAColor{
    gxColorValue  gray;
    gxColorValue  alpha;
};
```

- n **Indexed color value.** It contains an index value (of type `gxColorIndex`) and a reference to a color set object. The color is obtained by using the index value as an offset into the color set. Indexed color is defined by the `gxIndexedColor` type definition:

```
typedef long gxColorIndex;

struct gxIndexedColor{
    gxColorIndex  index;
    gxColorSet    set;
};
```

The Color Structure

A color value, plus a specification of the color space it belongs to, plus an optional reference to a color profile to use for color matching, constitute a color in QuickDraw GX. A color is a structure defined by the `gxColor` type definition:

```
struct gxColor{
    gxColorSpace      space;
    gxColorProfile    profile;
    union {
        struct gxCMYKColor    cmyk;
        struct gxRGBColor     rgb;
        struct gxRGBAColor    rgba;
        struct gxHSVColor     hsv;
        struct gxHLSColor     hls;
        struct gxXYZColor     xyz;
        struct gxYXYColor     yxy;
        struct gxLUVColor     luv;
        struct gxLABColor     lab;
        struct gxYIQColor     yiq;
        gxColorValue         gray;
        struct gxGrayAColor   graya;
        unsigned short       pixel16;
        unsigned long        pixel32;
        struct gxIndexedColor indexed;
        gxColorValue         component[4];
    } element;
};
```

Field descriptions

<code>space</code>	The color space for this color.
<code>profile</code>	A reference to a color profile to be used for color matching when drawing or when converting this color to another color space. If this field is <code>nil</code> , the default QuickDraw GX color profile is used for matching.
<code>element</code>	The color value for this color.

The `element` field is a union that can contain any one of the following fields:

<code>cmyk</code>	A CMYK color value.
<code>rgb</code>	An RGB color value.
<code>rgba</code>	An alpha-channel RGB color value.
<code>hsv</code>	An HSV color value.
<code>hls</code>	An HLS color value.
<code>xyz</code>	An XYZ color value.
<code>yxy</code>	A Yxy color value.

Colors and Color-Related Objects

luv	An L*u*v* color value.
lab	An L*a*b* color value.
yiiq	A YIQ color value.
gray	A grayscale color value
graya	An alpha-channel grayscale color value.
pixel16	A 16-bit pixel value, in <code>gxRGB16Space</code> format.
pixel32	A 32-bit pixel value, in any of the 32-bit color space formats.
indexed	An indexed color value.
component	An array of 4 undefined color-component values. Useful for indexing through the color one component at a time, as when working with different transfer modes for each color component.

Color Packing

You can store color values according to their standard definitions, or in packed format to save space. QuickDraw GX recognizes six kinds of color-value storage, defined in the `gxColorPackingTypes` enumeration:

```
enum gxColorPackingTypes{
    gxNoColorPacking      = 0x0000,
    gxAlphaSpace          = 0x0080,
    gxWord5ColorPacking   = 0x0500,
    gxLong8ColorPacking   = 0x0800,
    gxLong10ColorPacking  = 0x0a00,
    gxAlphaFirstPacking   = 0x1000
};
```

Constant descriptions

<code>gxNoColorPacking</code>	No packing applied; colors are stored with 16 bits per component.
<code>gxAlphaSpace</code>	An alpha channel is included in the color description. The alpha component follows the other components in storage.
<code>gxWord5ColorPacking</code>	Colors are stored with 5 bits per component. Unused bits in the storage space are the high-order bits.
<code>gxLong8ColorPacking</code>	Colors are stored with 8 bits per component. Unused bits in the storage space are the high-order bits.
<code>gxLong10ColorPacking</code>	Colors are stored with 10 bits per component. Unused bits in the storage space are the high-order bits.
<code>gxAlphaFirstPacking</code>	An alpha channel is included in the color description. The alpha component precedes the other components in storage.

The color-packing values are flags that are added to color-space definitions to define different kinds of packed color spaces. Note that the specification of an alpha channel in a color space is achieved with a color-packing flag. To see how these values are applied to the definitions of color spaces, see the section “Color Spaces,” next.

When QuickDraw GX converts from an unpacked color space to a packed color space, the color-component values are truncated (low-order bits lost) to fit the packed format. When QuickDraw GX converts from a packed color space to an unpacked color space, the color-component values are shifted leftward (padded with zeros in the low-order bits) to fit the unpacked format.

Color Spaces

A color space defines how a color value is represented. Each color space specifies the number, order, and size of the color-component values that make up a color value in that space. QuickDraw GX recognizes 31 color spaces, defined in the `gxColorSpace` enumeration:

```
enum gxColorSpaces{
    gxNoSpace          = 0,
    gxRGBSpace,
    gxCMYKSpace,
    gxHSVSpace,
    gxHLSpace,
    gxXYSpace,
    gxXYZSpace,
    gxLUVSpace,
    gxLABSpace,
    gxYIQSpace,
    gxNTSCSpace       = gxYIQSpace,
    gxPALSpace        = gxYIQSpace,
    gxGraySpace,
    gxIndexedSpace,
    gxRGBASpace       = gxRGBSpace + gxAlphaSpace,
    gxGrayASpace      = gxGraySpace + gxAlphaSpace,
    gxRGB16Space      = gxWord5ColorPacking + gxRGBSpace,
    gxRGB32Space      = gxLong8ColorPacking + gxRGBSpace,
    gxARGB32Space     = gxLong8ColorPacking + gxAlphaFirstPacking
                        + gxRGBASpace,
    gxCMYK32Space     = gxLong8ColorPacking + gxCMYKSpace,
    gxHSV32Space      = gxLong10ColorPacking + gxHSVSpace,
    gxHLS32Space      = gxLong10ColorPacking + gxHLSpace,
    gxXY32Space       = gxLong10ColorPacking + gxXYSpace,
    gxXYZ32Space      = gxLong10ColorPacking + gxXYZSpace,
    gxLUV32Space      = gxLong10ColorPacking + gxLUVSpace,
```

Colors and Color-Related Objects

```

    gxLAB32Space    = gxLong10ColorPacking + gxLABSpace,
    gxYIQ32Space    = gxLong10ColorPacking + gxYIQSpace,
    gxNTSC32Space   = gxYIQ32Space,
    gxPAL32Space    = gxYIQ32Space,
};

typedef long gxColorSpace;

```

Note that color spaces from `gxRGBASpace` through `gxYIQ32Space` use color-packing flags in their definitions. Those flags are described in the previous section, “Color Packing.”

The individual color spaces are described in the section “Color Spaces” beginning on page 4-6.

The Color Set Object

QuickDraw GX provides you with access to an individual color set object through a `gxColorSet` reference:

```
typedef struct gxPrivateColorSetRecord *gxColorSet;
```

In this type definition, `gxColorSet` is a type-checked reference, not an actual pointer to any defined structure. The contents of the color set object are private.

The gxSetColor Union

A color set object is essentially an array of color values. The acceptable types of color values that it may contain are defined by the `gxSetColor` union:

```

union gxSetColor{
    gxCMYKColor    cmyk;
    gxRGBColor     rgb;
    gxRGBAColor    rgba;
    gxHSVColor     hsv;
    gxHLSColor     hls;
    gxXYZColor     xyz;
    gxYXYColor     yxy;
    gxLUVColor     luv;
    gxLABColor     lab;
    gxYIQColor     yiq;
    gxColorValue   gray;
    gxGrayAColor   graya;
    unsigned short pixel16;
    unsigned long  pixel32;
    gxColorValue   component[4];
};

```

The `gxSetColor` union is an abbreviated `gxColor` structure. The `gxColor` structure is described on page 4-53.

The Color Profile Object

A color profile describes how to match a color with the colors in a color space. QuickDraw GX provides you with access to an individual color profile object through a `gxColorProfile` reference:

```
typedef struct gxPrivateProfileRecord *gxColorProfile;
```

In this type definition, `gxColorProfile` is a type-checked reference, not an actual pointer to any defined structure. The contents of the color profile object are private.

Color profile objects contain color profile structures as defined by the Macintosh ColorSync Utilities. See *Inside Macintosh: Advanced Color Imaging* for more information.

Color Functions

The functions in this section manipulate color structures, allowing you to test a color, compare two colors, combine two colors, and convert a color from one color space to another. Colors are described in the section “Color-Component Values, Color Values, and Colors” beginning on page 4-25. The color structure (type `gxColor`) is described on page 4-53.

GXCheckColor

You can use the `GXCheckColor` function to determine if a color is either within a given gamut in a particular color space, or representable in a given color set.

```
boolean GXCheckColor(const gxColor *source, gxColorSpace space,
                    gxColorSet aSet, gxColorProfile profile);
```

<code>source</code>	A pointer to the color to check.
<code>space</code>	The color space to check the source color against.
<code>aSet</code>	A reference to a color set to check the source color against. This parameter must be <code>nil</code> if the <code>space</code> parameter is not <code>gxIndexedSpace</code> .
<code>profile</code>	A reference to a color profile to check the source color against. <code>GXCheckColor</code> determines whether the source color is within the color gamut represented by this profile and the <code>space</code> color space.

function result `true` if the source color is contained in the specified color set, or if it is within the gamut of the specified color space and color profile; otherwise, `false`.

DESCRIPTION

The `GXCheckColor` function has two purposes. One is that you can use it to see if a given color exactly matches a color within a color set. For example, you can test whether a color matches a Pantone® or other spot color standard. To do this check, make sure that the `space` parameter specifies indexed color space and that the `aSet` parameter is not `nil`.

You can also use the `GXCheckColor` function to see if a given color can be drawn on a given view device. The function converts the source color to the color space represented in the `space` parameter, using the color profile in the `profile` parameter. If the resulting color is out of the gamut represented by `space` and `profile`, the function returns `false`.

SPECIAL CONSIDERATIONS

If you are using this function to test a color against a color set, it is unlikely to find a match (which must be exact) unless the source color and the color set referenced in the `aSet` parameter are based on the same color space and use identical color profiles.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`color_is_nil`
`colorSpace_out_of_range` (debugging version)

Warnings

`colorSet_index_out_of_range`

GXGetColorDistance

You can use the `GXGetColorDistance` function to determine the color-space distance between two colors.

```
Fixed GXGetColorDistance(const gxColor *target,
                        const gxColor *source);
```

`target` A pointer to the target color.
`source` A pointer to the source color.

function result The color-space distance between the two colors.

DESCRIPTION

The `GXGetColorDistance` function is useful in colorimetric applications and for judging perceived closeness of colors. It calculates how similar two colors are by determining the color-space distance between them. The distance calculation is performed in the color space of the target color. If the two colors are not in the same space, `GXGetColorDistance` converts the source color to the target color space before calculating the distance.

If the target color space is `gxIndexedSpace`, `GXGetColorDistance` uses the color space of the target color set.

The distance formula used is the standard Euclidean distance:

$$\text{distance} = \text{Sqrt}((b_0 - a_0)^2 + (b_1 - a_1)^2 + \dots);$$
SPECIAL CONSIDERATIONS

Because some of the color spaces are not linear, distances calculated in one space are not necessarily proportional to distances calculated in another space.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`color_is_nil`
`colorSpace_out_of_range` (debugging version)

Warnings

`colorSet_index_out_of_range`

GXCombineColor

You can use the `GXCombineColor` function to combine two colors with a transfer mode to get a result color for testing, without actually drawing.

```
gxColor *GXCombineColor(gxColor *target, gxInk operand);
```

`target` A pointer to the target color, which represents the destination color for drawing. On return, `target` points to the result color.

`operand` A reference to an ink object, which represents the source color and the transfer mode for drawing.

function result A pointer to the result color.

DESCRIPTION

The `GXCombineColor` function lets you preview or predict the results of drawing without actually carrying out a drawing operation. The function applies the color and transfer mode of the ink object referenced in the `operand` parameter to the color specified in the `target` parameter. It calculates the result of drawing, with the ink's color as the source color and the target color as the destination color.

`GXCombineColor` modifies the target color to reflect the operation and also returns a pointer to the resulting color. If `target` or `operand` is `nil`, the function posts an error and returns `nil`.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`ink_is_nil`
`color_is_nil`
`colorSpace_out_of_range` (debugging version)
`invalid_transferMode_colorSpace` (debugging version)

Warnings

`colorSet_index_out_of_range`

SEE ALSO

Ink objects and transfer modes for drawing are described in the chapter “Ink Objects” in this book.

GXConvertColor

You can use the `GXConvertColor` function to convert a color from one color space to another.

```
gxColor *GXConvertColor(gxColor *target, gxColorSpace space,
                        gxColorSet aSet, gxColorProfile profile);
```

`target` A pointer to the color to be converted. On return, `target` points to the converted color.

`space` The color space to convert the target color to.

`aSet` A reference to the color set to assign to the color space of the target color. This parameter must be `nil` if the `space` parameter is not `gxIndexedSpace`.

`profile` A reference to the color profile to assign to the converted color (that is, to use as the destination profile for the conversion). If you pass `nil` for this parameter, QuickDraw GX uses the default color profile.

function result A pointer to the converted color.

DESCRIPTION

The `GXConvertColor` function converts a color from one color space to another. The target color is both the input and the output color for this function; the function modifies the target color to reflect the conversion and also returns a pointer to the converted color. If `target` is `nil`, the function posts an error and returns `nil`.

If appropriate, `GXConvertColor` automatically performs color matching when converting the color. The color profile—if any—associated with the target color is used to correct the input color, and the color profile referenced in the `profile` parameter—if any—is used to create the final output color. If either color profile is `nil`, QuickDraw GX uses the default color profile in its place.

When converting to an indexed color space, `GXConvertColor` uses the color set specified by the `aSet` parameter as the color set for the returned color. It returns the closest existing color in the color set.

When converting from a color space without an alpha channel to one with an alpha channel, `GXConvertColor` gives the alpha channel value maximum opacity. When converting from a color space with an alpha channel to one without an alpha channel, the alpha-channel value is lost.

When converting from a color space with colors to a luminance-based (grayscale) color space, the color information is lost but `GXConvertColor` preserves luminance (overall lightness or brightness).

When converting between color spaces with different color packings (as from `gxRGB32Space` to `gxRGB16Space` or `gxRGBSpace`), `GXConvertColor` truncates or expands individual color-component values as appropriate.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`color_is_nil`
`colorSpace_out_of_range` (debugging version)

Warnings

`colorSet_index_out_of_range`

SEE ALSO

Color spaces are described in the section “Color Spaces” beginning on page 4-6. Color matching is described in the section “Color Conversion and Color Matching” beginning on page 4-26, and in the section “Converting and Matching Colors” beginning on page 4-41.

Color Set Functions

This section describes the functions with which you create color set objects, manipulate color set object properties, and retrieve and replace colors in a color set.

Creating and Manipulating Color Set Objects

The functions in this section allow you to create and manipulate color sets as QuickDraw GX objects.

GXGetDefaultColorSet

You can use the `GXGetDefaultColorSet` function to obtain a reference to the default color set object for a given pixel depth.

```
gxColorSet GXGetDefaultColorSet(long pixelDepth);
```

`pixelDepth` The pixel size of the color set.

function result A reference to the default color set with the specified pixel depth.

DESCRIPTION

Note that the return value of this function is a reference to the actual default color set object, not a copy of it. If you edit the color set returned by this function, you alter the actual default object that the system uses when creating new color set objects.

The valid values for `pixelDepth` are 1, 2, 4, and 8. Bitmaps with other pixel depths cannot use indexed color space.

You can also alter a default color set object using the `GXSetDefaultColorSet` function, described in the next section.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory  
invalid_pixelSize (debugging version)
```

SEE ALSO

Default color set objects are discussed in the section “Default Color Sets” on page 4-34. To modify a default color set object, use the `GXSetDefaultColorSet` function, described next.

To create a new color set object, use the `GXNewColorSet` function, described on page 4-64.

GXSetDefaultColorSet

You can use the `GXSetDefaultColorSet` function to replace the default color set object for a particular pixel depth.

```
void GXSetDefaultColorSet(gxColorSet target, long pixelDepth);
```

`target` A reference to the color set object to make the new default.

`pixelDepth` The pixel size of the color set.

DESCRIPTION

The `GXSetDefaultColorSet` function replaces an existing default color set with the color set specified by the `target` parameter. The pixel depth of the `target` color set determines which default color set is replaced.

This function disposes of the old default color set and increments the owner count of the new default color set.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`

`colorSet_is_nil`

`invalid_colorSet_count` (debugging version)

`invalid_pixelSize` (debugging version)

SEE ALSO

Default color set objects are discussed in the section “Default Color Sets” on page 4-34. To obtain a copy of a default color set object, use the `GXGetDefaultColorSet` function, described in the previous section.

To create a new color set object, use the `GXNewColorSet` function, described next.

GXNewColorSet

You can use the `GXNewColorSet` function to create a new color set object.

```
gxColorSet GXNewColorSet(gxColorSpace space, long count,
                          const gxSetColor colors[]);
```

<code>space</code>	The color space of the color set. You may not specify <code>gxIndexedSpace</code> for this parameter.
<code>count</code>	The size of the color space; the number of color values it contains.
<code>colors</code>	The array of color values that make up the color set.

function result A reference to the newly created color set object.

DESCRIPTION

The `GXNewColorSet` function creates a color set object with an owner count of 1 and returns a reference to it as the function result. You specify the number of colors in the color set in the `count` parameter, and pass the colors to the function in the `colors` array. Note that the array must contain color values of type `gxSetColor`.

You do not use this function to obtain a copy of a default color set; the `colors` array must contain one or more elements. If it does not, `GXNewColorSet` posts a `color_is_nil` error. If you specify `gxIndexedSpace` for the `space` parameter, this function posts a `colorSpace_out_of_range` error.

SPECIAL CONSIDERATIONS

If no error occurs, the `GXNewColorSet` function creates a color set object; you are responsible for disposing of that object when you no longer need it.

The current implementation of QuickDraw GX restricts the number of colors in a color set to a maximum of 65,535.

ERRORS, WARNINGS, AND NOTICES

Errors

<code>out_of_memory</code>	
<code>color_is_nil</code>	
<code>count_is_less_than_zero</code>	(debugging version)
<code>colorSpace_out_of_range</code>	(debugging version)
<code>number_of_colors_exceeds_implementation_limit</code>	

SEE ALSO

The `gxSetColor` union is described on page 4-56.

To obtain a copy of a default color set object, use the `GXGetDefaultColorSet` function, described on page 4-62.

GXDisposeColorSet

You can use the `GXDisposeColorSet` function to release a reference to a color set object.

```
void GXDisposeColorSet(gxColorSet target);
```

`target` A reference to the color set to dispose of.

DESCRIPTION

The `GXDisposeColorSet` function decrements the owner count of the color set specified by the `target` parameter and releases any memory used by the color set if the owner count goes to 0.

SPECIAL CONSIDERATIONS

If you attempt to dispose of a color set object used by an onscreen view device, this function posts a `colorSet_access_restricted` warning.

ERRORS, WARNINGS, AND NOTICES**Errors**

`colorSet_is_nil`

Warnings

`colorSet_access_restricted` (debugging version)

SEE ALSO

Owner counts are discussed in the section “Copying, Comparing, and Cloning Color Sets and Color Profiles” beginning on page 4-44, and in the section “Manipulating Owner Counts” beginning on page 4-46. To examine the owner count of a color set, use the `GXGetColorSetOwners` function, described on page 4-69.

GXCopyToColorSet

You can use the `GXCopyToColorSet` function to copy the contents of one existing color set to another, or to create a new color set and copy the contents of an existing color set into it.

```
gxColorSet GXCopyToColorSet(gxColorSet target, gxColorSet source);
```

`target` A reference to the color set to copy the source color set's contents into. If you specify `nil` for this parameter, the function creates a new color set.

`source` A reference to the color set whose contents you want to copy.

function result A reference to the color set copy.

DESCRIPTION

The `GXCopyToColorSet` function copies the contents of an existing color set object to another or it creates a new color set object and copies the contents of an existing color set object to it. The function copies the color space and color values and tag list (but not the owner count) of the color set object specified by the `source` parameter into the color set object specified by the `target` parameter. It clones, but does not copy, the tag objects in the tag list.

If you specify `nil` for the `target` parameter, `GXCopyToColorSet` creates a new color set object and copies the source color set's properties, including the owner count and tag list, into it.

You can use the `GXCopyToColorSet` function to create a copy of a color set and then modify it without changing the original.

SPECIAL CONSIDERATIONS

If you specify `nil` for the `target` parameter and no error occurs, the `GXCopyToColorSet` function creates a color set object; you are responsible for disposing of that object when you no longer need it.

If you specify a color set object used by an onscreen view device as the `target`, this function posts a `colorSet_access_restricted` warning.

ERRORS, WARNINGS, AND NOTICES**Errors**

out_of_memory
colorSet_is_nil

Warnings

colorSet_access_restricted (debugging version)

SEE ALSO

To create a new color set that is not a copy of an existing color set, use the `GXNewColorSet` function, described on page 4-64.

To compare two color set objects, use the `GXEqualColorSet` function, described in the next section.

GXEqualColorSet

You can use the `GXEqualColorSet` function to determine whether two color set objects are equal.

```
boolean GXEqualColorSet(gxColorSet one, gxColorSet two);
```

`one` A reference to one of the color sets to test for equality.

`two` A reference to the other color set to test for equality.

function result true if the two color sets are equal; false otherwise.

DESCRIPTION

The `GXEqualColorSet` function tests two color set objects for equality. For two color sets to be equal, they must have the same color space and identical color values—in the same order. Their owner counts and tag lists need not be identical.

ERRORS, WARNINGS, AND NOTICES**Errors**

out_of_memory
colorSet_is_nil

SEE ALSO

To make a copy of a color set object that is equal by the criteria of this function, use the `GXCopyToColorSet` function, described in the previous section.

GXCloneColorSet

You can use the `GXCloneColorSet` function to clone a color set—that is, to add a reference to it and increment its owner count.

```
gxColorSet GXCloneColorSet(gxColorSet source);
```

`source` A reference to the color set to clone.

function result A reference to the cloned color set.

DESCRIPTION

The `GXCloneColorSet` function increments the owner count of the color set referenced in the `source` parameter. You typically use this function when you want to create another reference to an existing color set rather than creating a distinct copy of the color set.

This function returns as its function result a reference to the color set—the same reference you pass in as the `source` parameter. It also increments the color set’s owner count.

SPECIAL CONSIDERATIONS

If you attempt to clone a color set object used by an onscreen view device, this function posts a `colorSet_access_restricted` warning.

ERRORS, WARNINGS, AND NOTICES

Errors

`colorSet_is_nil`

Warnings

`colorSet_access_restricted` (debugging version)

SEE ALSO

Owner counts for color set objects are discussed in the section “Copying, Comparing, and Cloning Color Sets and Color Profiles” beginning on page 4-44, and in the section “Manipulating Owner Counts” beginning on page 4-46.

To examine the owner count of a color set, use the `GXGetColorSetOwners` function, described on page 4-69. To decrement the owner count of a color set, use the `GXDisposeColorSet` function, described on page 4-65.

Manipulating Color Set Object Properties

The functions described in this section allow you to manipulate the common object properties of color sets: owner count and tag list. Functions for manipulating the colors in a color set are described in the section “Retrieving and Replacing Colors in a Color Set” beginning on page 4-73.

GXGetColorSetOwners

You can use the `GXGetColorSetOwners` function to determine the number of references to a particular color set object.

```
long GXGetColorSetOwners(gxColorSet source);
```

`source` A reference to the color set object to find the owner count of.

function result The owner count of the color set object referenced in the `source` parameter.

DESCRIPTION

The `GXGetColorSetOwners` function returns the owner count of the referenced color set. The owner count is the current number of references to the color set object.

ERRORS, WARNINGS, AND NOTICES

Errors

`colorSet_is_nil`

SEE ALSO

Owner counts for color set objects are discussed in the section “Copying, Comparing, and Cloning Color Sets and Color Profiles” beginning on page 4-44, and in the section “Manipulating Owner Counts” beginning on page 4-46.

GXGetColorSetTags

You can use the `GXGetColorSetTags` function to examine one or more of the tag objects associated with a color set object.

```
long GXGetColorSetTags(gxColorSet source, long tagType,
                       long index, long count, gxTag items[]);
```

<code>source</code>	A reference to the color set object to examine the tag list of.
<code>tagType</code>	The type of tag object to search for. A value of 0 indicates that you want to look for all tag types.
<code>index</code>	The (1-based) index of the first such tag reference to return.
<code>count</code>	The number of tag references to return
<code>items</code>	An array to hold the returned tag references.

function result The number of tag references found that fit the criteria.

DESCRIPTION

The `GXGetColorSetTags` function searches the tag list of the `source` color set object for references to tag objects with the tag type specified by the `tagType` parameter. If you specify 0 for the `tagType` parameter, the `GXGetColorSetTags` function searches all tag types.

You can use the `index` and the `count` parameters to specify which tag references of the appropriate type the `GXGetColorSetTags` function should return. The `index` parameter indicates the first tag reference to return and the `count` parameter indicates how many tag references to return. The `index` parameter must be greater than 0. The `count` parameter must be greater than 0 or equal to the `gxSelectToEnd` constant (-1), which indicates that all tag references (starting with the tag reference indicated by the `index` parameter) should be returned.

If you pass a value other than `nil` for the `items` parameter, the `GXGetColorSetTags` function returns in it the tag references that were found. Regardless of the value you pass for `items`, the function result is the number of tag references found that fit the criteria.

ERRORS, WARNINGS, AND NOTICES**Errors**

```

out_of_memory
colorSet_is_nil
index_is_less_than_one    (debugging version)
count_is_less_than_one    (debugging version)

```

Warnings

```

index_out_of_range
count_out_of_range

```

SEE ALSO

Tag objects are discussed in the chapter “Tag Objects” in this book.

To change the set of tag references associated with a color set object, use the `GXSetColorSetTags` function, described in the next section.

GXSetColorSetTags

You can use the `GXSetColorSetTags` function to add, remove, or replace tag objects associated with a color set object.

```

void GXSetColorSetTags(gxColorSet target, long tagType,
                       long index, long oldCount,
                       long newCount, const gxTag items[]);

```

<code>target</code>	A reference to the color set object to alter the tag list of.
<code>tagType</code>	The type of tag objects to replace. A value of 0 indicates that you want to replace tags of all types.
<code>index</code>	The (1-based) index of the first tag reference (to a tag object of the appropriate type) to replace.
<code>oldCount</code>	The number of tag references to replace. A value of 0 specifies that you want to insert tag references before the tag reference indicated by the <code>index</code> parameter, rather than replace tag references. A value of -1 (the <code>gxSelectToEnd</code> constant) specifies that all tag references of the requested type, starting with the tag reference indicated by the <code>index</code> parameter, should be replaced.
<code>newCount</code>	The number of tag references to insert. A value of 0 specifies that there are no tag references to insert; the existing tag references that match the criteria you specify are removed from the source color set's tag list and disposed of.
<code>items</code>	An array of tag references to insert in the tag list.

DESCRIPTION

The `GXSetColorSetTags` function allows you add tag references to a color set object's tag list, to remove tag references from the list, or to replace tag references in the list with new tag references. In any of these three cases, the `target` parameter specifies the color set object to be modified, the `newCount` parameter specifies the number of tag references to add, and the `items` parameter provides the new tag references.

- n To add tag references, set the `oldCount` parameter to 0. Use the `tagType` and the `index` parameters to specify where to add the new tag references. (For example, if you specify `nil` for the `tagType` parameter and 1 for the `index` parameter, this function inserts the new tag references before the current tag references. If you specify a value other than `nil` for the `tagType` parameter and a value of 2 for the `index` parameter, the function inserts the new tag references before the second tag reference with a tag type matching the `tagType` parameter.)
- n To remove tag references, set the `newCount` parameter to 0 and the `items` parameter to `nil`. You can use the `index` and the `oldCount` parameters to specify which tag references of the specified type should be removed. The `index` parameter indicates the first tag reference of the specified type to remove and the `oldCount` parameter indicates how many tag references of the specified type to remove.
- n To replace tag references, use the `tagType`, `index`, and `oldCount` parameters to indicate which tag references to replace, and use the `newCount` and `items` parameters to specify the new tag references to add. If `newCount` is greater than `oldCount`, the extra tag references are placed immediately adjacent to the last tag reference replaced.

SPECIAL CONSIDERATIONS

If you attempt to modify the tag list of a color set object used by an onscreen view device, this function posts a `colorSet_access_restricted` warning.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>out_of_memory</code>	
<code>colorSet_is_nil</code>	
<code>tag_is_nil</code>	
<code>parameter_is_nil</code>	(debugging version)
<code>inconsistent_parameters</code>	(debugging version)
<code>parameter_out_of_range</code>	(debugging version)
<code>index_is_less_than_zero</code>	(debugging version)
<code>cannot_dispose_locked_tag</code>	(debugging version)

Warnings

<code>index_out_of_range</code>	
<code>count_out_of_range</code>	
<code>colorSet_access_restricted</code>	(debugging version)

Notices (debugging version)

<code>tag_already_set</code>	
------------------------------	--

SEE ALSO

Tag objects are discussed in the chapter “Tag Objects” in this book.

To examine the set of tag references associated with a color set object, use the `GXGetColorSetTags` function, described in the previous section.

Retrieving and Replacing Colors in a Color Set

The functions described in this section allow you to manipulate the colors in a color set. Functions for manipulating the other properties of color sets are described in the section “Manipulating Color Set Object Properties” beginning on page 4-69.

GXGetColorSet

You can use the `GXGetColorSet` function to retrieve the color values from a color set object.

```
long GXGetColorSet(gxColorSet source, gxColorSpace *space,
                  gxSetColor colors[]);
```

<code>source</code>	A reference to the color set object whose color values you want to retrieve.
<code>space</code>	A pointer to a color space value. On return, specifies the color space for the source color set.
<code>colors</code>	An array of <code>gxSetColor</code> color values. On return, contains the set of color values in the source color set.

function result The number of color values in the source color set.

DESCRIPTION

The `GXGetColorSet` function retrieves the color values from the source color set and returns them in the `colors` array. It also returns the color set’s color space in the location pointed to by the `space` parameter. The function result is the number of colors returned in the `colors` array.

Before calling `GXGetColorSet`, you must allocate an array of sufficient size to hold the color-value array of the color set. If instead you pass `nil` for the `colors` parameter, the function does not return any color values, but nonetheless returns (as its function result) the number of colors in the color set. Thus you can make an initial call to `GXGetColorSet` to determine the size of the array to allocate, and then call it once more to get the color values themselves.

ERRORS, WARNINGS, AND NOTICES**Errors**

out_of_memory
colorSet_is_nil

SEE ALSO

To replace the entire array of color values in a color set object, use the `GXSetColorSet` function, described in the next section. To retrieve some of the color values in a color set object, use the `GXGetColorSetParts` function, described on page 4-75. To replace some of the color values in a color set object, use the `GXSetColorSetParts` function, described on page 4-76.

The `gxSetColor` union is described on page 4-56.

GXSetColorSet

You can use the `GXSetColorSet` function to replace the color values of a color set object.

```
void GXSetColorSet(gxColorSet target, gxColorSpace space,
                  long count, const gxSetColor colors[]);
```

<code>target</code>	A reference to the color set object whose color values you want to replace.
<code>space</code>	The new color space for the color set referenced in the <code>target</code> parameter.
<code>count</code>	The number of color values in the <code>colors</code> array.
<code>colors</code>	The array of color values to assign to the color set.

DESCRIPTION

The `GXSetColorSet` function assigns the specified color space and color values to the target color set. If `gxNoSpace` is passed in the `space` parameter, the color space is unchanged. If the `colors` array is `nil` and if `count` is zero, the color set remains unchanged.

SPECIAL CONSIDERATIONS

If you attempt to modify the color values of a color set object used by an onscreen view device, this function posts a `colorSet_access_restricted` warning.

The current implementation of QuickDraw GX restricts the number of colors in a color set to a maximum of 65,535.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>out_of_memory</code>	
<code>colorSet_is_nil</code>	
<code>inconsistent_parameters</code>	(debugging version)
<code>count_is_less_than_zero</code>	(debugging version)
<code>colorSpace_out_of_range</code>	(debugging version)
<code>number_of_colors_exceeds_implementation_limit</code>	

Warnings

<code>colorSet_access_restricted</code>	(debugging version)
---	---------------------

SEE ALSO

To retrieve the entire array of color values from a color set object, use the `GXGetColorSet` function, described in the previous section. To retrieve some of the color values in a color set object, use the `GXGetColorSetParts` function, described in the next section. To replace some of the color values in a color set object, use the `GXSetColorSetParts` function, described on page 4-76.

The `gxSetColor` union is described on page 4-56.

GXGetColorSetParts

You can use the `GXGetColorSetParts` function to retrieve specified colors from a color set object.

```
long GXGetColorSetParts(gxColorSet source, long index, long count,
                       gxColorSpace *space, gxSetColor data[]);
```

<code>source</code>	A reference to the color set object whose color values you want to retrieve.
<code>index</code>	The first color value to retrieve. To retrieve the first color value in the color set, specify 1 for this parameter.
<code>count</code>	The number of color values to retrieve. Specify <code>gxSelectToEnd</code> to retrieve all color values in the color set including and beyond <code>index</code> .
<code>space</code>	A pointer to a color space value. On return, specifies the color space for the source color set.
<code>data</code>	An array of <code>gxSetColor</code> color values. On return, contains the specified subset of color values from the source color set.

function result The number of color values in the range specified by `index` and `count`.

DESCRIPTION

The `GXGetColorSetParts` function retrieves the specified color values from the source color set and returns them in the `data` array. It also returns the color set's color space in the location pointed to by the `space` parameter. The function result is the number of color values copied into the `data` array.

Before calling `GXGetColorSetParts`, you must allocate an array of sufficient size to hold the specified number of color values. If instead you pass `nil` for the `data` parameter, the function does not return any color values, but nonetheless returns (as its function result) the number of colors in the specified range. Thus you can make an initial call to `GXGetColorSetParts` to determine the size of the array to allocate, and then call it once more to get the color values themselves.

ERRORS, WARNINGS, AND NOTICES**Errors**

```
out_of_memory
colorSet_is_nil
index_is_less_than_one      (debugging version)
count_is_less_than_one     (debugging version)
```

Warnings

```
index_out_of_range
count_out_of_range
```

SEE ALSO

To retrieve the entire array of color values from a color set object, use the `GXGetColorSet` function, described on page 4-73. To replace the entire array of color values in a color set object, use the `GXSetColorSet` function, described in the previous section. To replace some of the color values in a color set object, use the `GXSetColorSetParts` function, described in the next section.

The `gxSetColor` union is described on page 4-56.

GXSetColorSetParts

You can use the `GXSetColorSetParts` function to replace specified colors in a color set object.

```
void GXSetColorSetParts(gxColorSet target, long index,
                        long oldCount, long newCount,
                        const gxSetColor data[]);
```

`target` A reference to the color set object whose color values you want to modify.
`index` The first color value to replace. To replace the first color value in the color set, specify 1 for this parameter.

Colors and Color-Related Objects

<code>oldCount</code>	The number of color values to replace. Specify <code>gxSelectToEnd</code> to replace all color values in the color set including and beyond <code>index</code> .
<code>newCount</code>	The number of new color values to add; that is, the number of color values in the <code>data</code> array.
<code>data</code>	The array of color values to add to the color set.

DESCRIPTION

The `GXSetColorSetParts` function assigns the specified color values to the target color set, starting at the location specified by `index` after first removing the number of existing color values specified by `oldCount`.

This function does not accept the `gxSetToNil` constant for the `data` parameter. If you want to simply remove colors, pass 0 for `newCount`.

The current implementation of QuickDraw GX restricts the number of colors in a color set to a maximum of 65,535.

SPECIAL CONSIDERATIONS

If you attempt to modify the color values of a color set object used by an onscreen view device, this function posts a `colorSet_access_restricted` warning.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>out_of_memory</code>	
<code>colorSet_is_nil</code>	
<code>inconsistent_parameters</code>	(debugging version)
<code>index_is_less_than_zero</code>	(debugging version)
<code>count_is_less_than_zero</code>	(debugging version)
<code>number_of_colors_exceeds_implementation_limit</code>	

Warnings

<code>index_out_of_range</code>	
<code>count_out_of_range</code>	
<code>colorSet_access_restricted</code>	(debugging version)

SEE ALSO

To retrieve the entire array of color values from a color set object, use the `GXGetColorSet` function, described on page 4-73. To replace the entire array of color values in a color set object, use the `GXSetColorSet` function, described on page 4-74. To retrieve some of the color values in a color set object, use the `GXGetColorSetParts` function, described in the previous section.

The `gxSetColor` union is described on page 4-56.

Color Profile Functions

This section describes the functions with which you create color profile objects, manipulate color profile object properties, and retrieve and replace profile information.

Creating and Manipulating Color Profile Objects

The functions in this section allow you to create and manipulate color profiles as QuickDraw GX objects. For descriptions of functions that manipulate the properties of color profile objects, see the sections “Manipulating Color Profile Object Properties” beginning on page 4-84 and “Retrieving and Replacing Profile Information” beginning on page 4-88.

GXGetDefaultColorProfile

You can use the `GXGetDefaultColorProfile` function to obtain a reference to the default color profile object.

```
gxColorProfile GXGetDefaultColorProfile(void);
```

function result A reference to the default color profile.

DESCRIPTION

The default color profile is the color profile for the Apple 13-inch color monitor. When converting or matching colors, QuickDraw GX assumes the default color profile for any color, bitmap, or transfer mode whose color profile property is `nil`.

Note that the return value of this function is a reference to the actual default color profile object, not a copy of it. You should not make changes to the profile; if you edit it (for example, by calling `GXLockProfile` and `GXGetProfileStructure`), you alter the actual default profile that QuickDraw GX uses when creating new color profile objects.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`

SEE ALSO

The default color profile object is discussed in the section “The Default Color Profile” beginning on page 4-37.

To create a copy of the default color profile object, you can use the `GXCopyToColorProfile` function, described on page 4-81.

To create a new color profile, use the `GXNewColorProfile` function, described next.

GXNewColorProfile

You can use the `GXNewColorProfile` function to create a new color profile object.

```
gxColorProfile GXNewColorProfile(long size,
                                  void *colorProfileData);
```

`size` The size in bytes of the profile data to assign to the new color profile object.

`colorProfileData` A pointer to the profile data to assign to the new color profile object.

function result A reference to the newly created color profile object.

DESCRIPTION

The `GXNewColorProfile` function creates a color profile object with an owner count of 1 from the profile data that you supply. The new color profile object is not a copy of the default color profile.

If you specify a nonzero value for the `size` parameter, you must pass a `ColorSync` color profile structure to `GXNewColorProfile`. The function does not check for the validity of the profile data, but if the `colorProfileData` parameter is `nil` and the `size` parameter is nonzero the function posts an error.

You can create a zero-length profile by passing 0 for the `size` parameter when calling this function. The effect of a zero-length profile is to inhibit color matching.

SPECIAL CONSIDERATIONS

If no error occurs, the `GXNewColorProfile` function creates a color profile object; you are responsible for disposing of that object when you no longer need it.

ERRORS, WARNINGS, AND NOTICES**Errors**

out_of_memory
 parameter_is_nil (debugging version)

SEE ALSO

Zero-length profiles are described in the section “Zero-Length Profiles” on page 4-37.

The format of the profile data in a color profile object is described in the section “Profile Data” beginning on page 4-36. The ColorSync Utilities are described in *Inside Macintosh: Advanced Color Imaging*.

To obtain a reference to the default color profile, use the `GXGetDefaultColorProfile` function, described in the previous section.

GXDisposeColorProfile

You can use the `GXDisposeColorProfile` function to release a reference to a color profile object.

```
void GXDisposeColorProfile(gxColorProfile target);
```

target A reference to the color profile to dispose of.

DESCRIPTION

The `GXDisposeColorProfile` function decrements the owner count of the color profile object referenced in the `target` parameter and releases any memory used by the color profile if the owner count goes to 0.

SPECIAL CONSIDERATIONS

If you attempt to dispose of a color profile object used by an onscreen view device, this function posts a `colorProfile_access_restricted` warning.

ERRORS, WARNINGS, AND NOTICES**Errors**

colorProfile_is_nil

Warnings

colorProfile_access_restricted (debugging version)

SEE ALSO

Owner counts are discussed in the section “Copying, Comparing, and Cloning Color Sets and Color Profiles” beginning on page 4-44, and in the section “Manipulating Owner Counts” beginning on page 4-46. To examine the owner count of a color profile, use the `GXGetColorProfileOwners` function, described on page 4-84.

GXCopyToColorProfile

You can use the `GXCopyToColorProfile` function to copy the contents of an existing color profile object into another or to create a new color profile object and copy the contents of an existing color profile into it.

```
gxColorProfile GXCopyToColorProfile(gxColorProfile target,
                                     gxColorProfile source);
```

target A reference to the color profile to copy the source contents into. If you specify `nil` for this parameter, the `GXCopyToColorProfile` function creates a new color profile.

source A reference to the color profile object whose contents you want to copy.

function result A reference to the color profile copy.

DESCRIPTION

The `GXCopyToColorProfile` function either copies the contents of an existing color profile object to another or creates a new color profile object and copies the contents of an existing color profile object to it. The function copies the profile data and tag list (but not the owner count) of the color profile specified by the `source` parameter into the color profile specified by the `target` parameter. It clones, but does not copy, the tag objects in the tag list.

If you specify `nil` for the `target` parameter, `GXCopyToColorProfile` creates a new color profile object and copies the source properties, including the owner count and tag list, into it.

You can use the `GXCopyToColorProfile` function to create a copy of a color profile and then modify it without changing the original.

SPECIAL CONSIDERATIONS

If you specify `nil` for the `target` parameter and no error occurs, the `GXCopyToColorProfile` function creates a color profile object; you are responsible for disposing of that object when you no longer need it.

If you specify a color profile object used by an onscreen view device as the `target`, this function posts a `colorProfile_access_restricted` warning.

ERRORS, WARNINGS, AND NOTICES**Errors**

out_of_memory
colorProfile_is_nil

Warnings

colorProfile_access_restricted (debugging version)

SEE ALSO

To create a new color profile that is not a copy of an existing color profile, use the `GXNewColorProfile` function, described on page 4-79.

To compare two color profile objects, use the `GXEqualColorProfile` function, described next.

GXEqualColorProfile

You can use the `GXEqualColorProfile` function to determine whether two color profile objects are equal.

```
boolean GXEqualColorProfile(gxColorProfile one,
                             gxColorProfile two);
```

`one` A reference to one of the color profiles to test for equality.

`two` A reference to the other color profile to test for equality.

function result true if the color profiles are equal; false otherwise.

DESCRIPTION

The `GXEqualColorProfile` function tests two color profile objects for equality. For two color profiles to be equal, they must have exactly the same profile data, although their owner counts and tag lists need not be identical.

ERRORS, WARNINGS, AND NOTICES**Errors**

out_of_memory
colorProfile_is_nil

SEE ALSO

To make a copy of a color profile object that is equal by the criteria of this function, use the `GXCopyToColorProfile` function, described in the previous section.

GXCloneColorProfile

You can use the `GXCloneColorProfile` function to clone a color profile—that is, to add a reference to it and increment its owner count.

```
gxColorProfile GXCloneColorProfile(gxColorProfile source);
```

`source` A reference to the color profile to clone.

function result A reference to the cloned color profile.

DESCRIPTION

The `GXCloneColorProfile` function increments the owner count of the color profile referenced in the `source` parameter. You typically use this function when you want to create another reference to an existing color profile rather than creating a distinct copy of the color profile.

This function returns as its function result a reference to the color profile—the same reference you pass in as the `source` parameter. It also increments the color profile's owner count.

SPECIAL CONSIDERATIONS

If you attempt to clone a color profile object used by an onscreen view device, this function posts a `colorProfile_access_restricted` warning.

ERRORS, WARNINGS, AND NOTICES

Errors

`colorProfile_is_nil`

Warnings

`colorProfile_access_restricted` (debugging version)

SEE ALSO

Owner counts for color profile objects are discussed in the section “Copying, Comparing, and Cloning Color Sets and Color Profiles” beginning on page 4-44, and in the section “Manipulating Owner Counts” beginning on page 4-46.

To examine the owner count of a color profile, use the `GXGetColorProfileOwners` function, described on page 4-84. To decrement the owner count of a color profile, use the `GXDisposeColorProfile` function, described on page 4-80.

Manipulating Color Profile Object Properties

The functions described in this section allow you to manipulate the common object properties of color profile objects: owner count and tag list. For descriptions of functions that manipulate the profile data of color profile objects, see the section “Retrieving and Replacing Profile Information” beginning on page 4-88. For descriptions of functions that allow you to create and manipulate color profiles as QuickDraw GX objects, see the section “Creating and Manipulating Color Profile Objects” beginning on page 4-78.

GXGetColorProfileOwners

You can use the `GXGetColorProfileOwners` function to determine the number of references to a particular color profile object.

```
long GXGetColorProfileOwners(gxColorProfile source);
```

`source` A reference to the color profile object to find the owner count of.

function result The owner count of the source color profile object.

DESCRIPTION

The `GXGetColorProfileOwners` function returns the owner count of the referenced color profile object. The owner count is the current number of references to the color profile object.

ERRORS, WARNINGS, AND NOTICES

Errors

`colorProfile_is_nil`

SEE ALSO

Owner counts for color profile objects are discussed in the section “Copying, Comparing, and Cloning Color Sets and Color Profiles” beginning on page 4-44, and in the section “Manipulating Owner Counts” beginning on page 4-46.

GXGetColorProfileTags

You can use the `GXGetColorProfileTags` function to examine one or more of the tag objects associated with a color profile object.

```
long GXGetColorProfileTags(gxColorProfile source, long tagType,
                           long index, long count, gxTag items[]);
```

<code>source</code>	A reference to the color profile object to examine the tag list of.
<code>tagType</code>	The type of tag object to search for. A value of 0 indicates that you want to look for all tag types.
<code>index</code>	The (1-based) index of the first such tag reference to return.
<code>count</code>	The number of tag references to return.
<code>items</code>	An array to hold the returned tag references.

function result The number of tag references found that fit the criteria.

DESCRIPTION

The `GXGetColorProfileTags` function searches the tag list of the `source` color profile object for references to tag objects with the tag type specified by the `tagType` parameter. If you specify 0 for the `tagType` parameter, the `GXGetColorProfileTags` function searches all tag types.

You can use the `index` and the `count` parameters to specify which tag references of the appropriate type the `GXGetColorProfileTags` function should return. The `index` parameter indicates the first tag reference to return and the `count` parameter indicates how many tag references to return. The `index` parameter must be greater than 0. The `count` parameter must be greater than 0 or equal to the `gxSelectToEnd` constant (-1), which indicates that all tag references (starting with the tag reference indicated by the `index` parameter) should be returned.

If you pass a value other than `nil` for the `items` parameter, the `GXGetColorProfileTags` function returns in it the tag references that were found. Regardless of the value you pass for `items`, the function result is the number of tag references found that fit the criteria.

ERRORS, WARNINGS, AND NOTICES

Errors

<code>out_of_memory</code>	
<code>colorProfile_is_nil</code>	
<code>index_is_less_than_one</code>	(debugging version)
<code>count_is_less_than_one</code>	(debugging version)

Warnings

<code>index_out_of_range</code>
<code>count_out_of_range</code>

SEE ALSO

Tag objects are discussed in the chapter “Tag Objects” in this book.

To change the set of tag references associated with a color profile object, use the `GXSetColorProfileTags` function, described in the next section.

GXSetColorProfileTags

You can use the `GXSetColorProfileTags` function to add, remove, or replace tag objects associated with a color profile object.

```
void GXSetColorProfileTags(gxColorProfile target, long tagType,
                          long index, long oldCount,
                          long newCount, const gxTag items[]);
```

<code>target</code>	A reference to the color profile object to alter the tag list of.
<code>tagType</code>	The type of tag objects to replace. A value of 0 indicates that you want to replace tags of all types.
<code>index</code>	The (1-based) index of the first tag reference (to a tag object of the appropriate type) to replace.
<code>oldCount</code>	The number of tag references to replace. A value of 0 specifies that you want to insert tag references before the tag reference indicated by the <code>index</code> parameter, rather than replace tag references. A value of -1 (the <code>gxSelectToEnd</code> constant) specifies that all tag references of the requested type, starting with the tag reference indicated by the <code>index</code> parameter, should be replaced.
<code>newCount</code>	The number of tag references to insert. A value of 0 specifies that there are no tag references to insert; the existing tag references that match the criteria you specify are removed from the source color profile’s tag list and disposed of.
<code>items</code>	An array of tag references to insert in the tag list.

DESCRIPTION

The `GXSetColorProfileTags` function allows you add tag references to a color profile object’s tag list, to remove tag references from the list, or to replace tag references in the list with new tag references. In any of these three cases, the `target` parameter specifies the color profile object to be modified, the `newCount` parameter specifies the number of tag references to add, and the `items` parameter provides the new tag references.

Colors and Color-Related Objects

- n To add tag references, set the `oldCount` parameter to 0. Use the `tagType` and the `index` parameters to specify where to add the new tag references. (For example, if you specify `nil` for the `tagType` parameter and 1 for the `index` parameter, this function inserts the new tag references before the current tag references. If you specify a value other than `nil` for the `tagType` parameter and a value of 2 for the `index` parameter, the function inserts the new tag references before the second tag reference with a tag type matching the `tagType` parameter.)
- n To remove tag references, set the `newCount` parameter to 0 and the `items` parameter to `nil`. You can use the `index` and the `oldCount` parameters to specify which tag references of the specified type should be removed. The `index` parameter indicates the first tag reference of the specified type to remove and the `oldCount` parameter indicates how many tag references of the specified type to remove.
- n To replace tag references, use the `tagType`, `index`, and `oldCount` parameters to indicate which tag references to replace, and use the `newCount` and `items` parameters to specify the new tag references to add. If `newCount` is greater than `oldCount`, the extra tag references are placed immediately adjacent to the last tag reference replaced.

SPECIAL CONSIDERATIONS

If you attempt to modify the tag list of a color profile object used by an onscreen view device, this function posts a `colorProfile_access_restricted` warning.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>out_of_memory</code>	
<code>colorSet_is_nil</code>	
<code>tag_is_nil</code>	
<code>parameter_is_nil</code>	(debugging version)
<code>inconsistent_parameters</code>	(debugging version)
<code>parameter_out_of_range</code>	(debugging version)
<code>index_is_less_than_zero</code>	(debugging version)
<code>cannot_dispose_locked_tag</code>	(debugging version)

Warnings

<code>index_out_of_range</code>	
<code>count_out_of_range</code>	
<code>colorProfile_access_restricted</code>	(debugging version)

Notices (debugging version)

<code>tag_already_set</code>	
------------------------------	--

SEE ALSO

Tag objects are discussed in the chapter “Tag Objects” in this book.

To examine the set of tag references associated with a color profile object, use the `GXGetColorProfileTags` function, described in the previous section.

Retrieving and Replacing Profile Information

The functions described in this section allow you to manipulate the profile data of color profile objects. For descriptions of functions that manipulate the common object properties of color profile object, see the section “Manipulating Color Profile Object Properties” beginning on page 4-84. For descriptions of functions that allow you to create and manipulate color profiles as QuickDraw GX objects, see the section “Creating and Manipulating Color Profile Objects” beginning on page 4-78.

GXGetColorProfile

You can use the `GXGetColorProfile` function to retrieve the profile data from a color profile object.

```
long GXGetColorProfile(gxColorProfile source,
                      void *colorProfileData);
```

`source` A reference to the color profile object to get the profile data from.

`colorProfileData` A pointer to a buffer. On return, the buffer contains the profile data for the source color profile.

function result The size in bytes of the source color profile’s profile data.

DESCRIPTION

The `GXGetColorProfile` function returns the profile data from the source color profile in the buffer pointed to by the `responses` parameter. It also returns the size of the profile data as the function result.

The profile data returned by this function is a ColorSync color profile structure (type `CMProfile`).

If you specify `nil` for the `colorProfileData` parameter, this function does not return the profile data, but it nevertheless returns a correct value for the size of the profile response structure in the function result. Thus you can make an initial call to `GXGetColorProfile` to determine the size of buffer to allocate, and then call it once more to get the profile data itself.

ERRORS, WARNINGS, AND NOTICES**Errors**

out_of_memory
colorProfile_is_nil

SEE ALSO

To replace the profile data in a color profile object, use the `GXSetColorProfile` function, described in the next section.

The format of the profile data in a color profile object is described in the section “Profile Data” beginning on page 4-36. The ColorSync Utilities, including the `CMProfile` data type, are described in *Inside Macintosh: Advanced Color Imaging*.

GXSetColorProfile

You can use the `GXSetColorProfile` function to assign profile data to a color profile object.

```
void GXSetColorProfile(gxColorProfile target, long size,
                      void *colorProfileData);
```

<code>target</code>	A reference to the color profile object whose profile data you want to change.
<code>size</code>	The size in bytes of the profile data to assign to the target color profile.
<code>colorProfileData</code>	A pointer to the profile data.

DESCRIPTION

The `GXSetColorProfile` function assigns the specified profile data to the target color profile. If you specify a nonzero value for the `size` parameter, the pointer to the profile data must not be `nil`. It should be in the form of a valid ColorSync color profile structure (type `CMProfile`), although the function does not actually verify this.

If you pass 0 for the `size` parameter to this function, QuickDraw GX converts this profile into a zero-length profile, which you can use to inhibit color matching.

SPECIAL CONSIDERATIONS

If you attempt to alter the profile data of a color profile object used by an onscreen view device, this function posts a `colorProfile_access_restricted` warning.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>out_of_memory</code>	
<code>colorProfile_is_nil</code>	
<code>inconsistent_parameters</code>	(debugging version)
<code>parameter_out_of_range</code>	(debugging version)

Warnings

<code>colorProfile_access_restricted</code>	(debugging version)
---	---------------------

SEE ALSO

To examine the profile data in a color profile object, use the `GXGetColorProfile` function, described in the previous section.

Zero-length profiles are described in the section “Zero-Length Profiles” on page 4-37.

The format of the profile data in a color profile object is described in the section “Profile Data” beginning on page 4-36. The ColorSync Utilities, including the `CMProfile` data type, are described in *Inside Macintosh: Advanced Color Imaging*.

GXLockColorProfile

You can use the `GXLockColorProfile` function to load a color profile object into memory and lock its profile data into a fixed memory location.

```
void GXLockColorProfile (gxColorProfile source);
```

`source` A reference to the color profile to be loaded and locked.

DESCRIPTION

The `GXLockColorProfile` function prevents a color profile from being relocated, so that you can manipulate its profile data directly in QuickDraw GX memory rather than working with a copy of it in application memory.

To directly edit the color profile, call `GXLockColorProfile` followed by `GXGetColorProfileStructure`; after editing, call `GXUnlockColorProfile`.

SPECIAL CONSIDERATIONS

To avoid fragmenting the QuickDraw GX heap, call the `GXUnlockColorProfile` function as soon as possible after calling `GXLockColorProfile`.

In low memory situations with a fragmented heap, QuickDraw GX can unlock locked objects without warning. Be careful about making memory-intensive calls when you are working with a locked color profile.

ERRORS, WARNINGS, AND NOTICES**Errors**

out_of_memory
colorProfile_is_nil

SEE ALSO

The `GXUnlockColorProfile` and `GXGetColorProfileStructure` functions are described in the next two sections.

GXUnlockColorProfile

You can use the `GXUnlockColorProfile` function to allow QuickDraw GX to relocate, compress, or unload a color profile object that has been locked.

```
void GXUnlockColorProfile (gxColorProfile source);
```

gxColorProfile

A reference to the color profile to be unlocked.

DESCRIPTION

To directly edit the color profile, call `GXLockColorProfile` followed by `GXGetColorProfileStructure`; after editing, call `GXUnlockColorProfile`.

Once you call `GXUnlockColorProfile`, the profile data may be relocated and a pointer returned by `GXGetColorProfileStructure` may no longer be valid.

SPECIAL CONSIDERATIONS

To avoid fragmenting the QuickDraw GX heap, call the `GXUnlockColorProfile` function as soon as possible after calling `GXLockColorProfile`.

ERRORS, WARNINGS, AND NOTICES**Errors**

colorProfile_is_nil

SEE ALSO

The `GXLockColorProfile` function is described in the previous section. The `GXGetColorProfileStructure` function is described next.

GXGetColorProfileStructure

You can use the `GXGetColorProfileStructure` function to get a pointer to the profile data of a color profile object.

```
void *GXGetColorProfileStructure(gxColorProfile source,
                                long *length);
```

`source` A reference to the color profile object whose profile data you need access to.

`length` A pointer to a `long` value. On return, the value specifies the size in bytes of the profile data.

function result A pointer to the profile data of the source color profile.

DESCRIPTION

The `GXGetColorProfileStructure` function determines the size of the profile data in a color profile object and returns a pointer to the data in the QuickDraw GX heap. You can use the pointer to examine or change the profile data without copying the data into your application's heap and back again.

Before calling this function, call `GXLockColorProfile` to lock the profile data in memory; after editing the profile data, call `GXUnlockColorProfile` to free the profile data for relocation.

The profile data returned by this function is a `ColorSync` color profile structure (type `CMPProfile`).

This function is useful even if you do not intend to edit a color profile. You can use it to simply read a specific piece of color profile data, such as the white point, without having to obtain a copy of the entire profile.

SPECIAL CONSIDERATIONS

To avoid fragmenting the QuickDraw GX heap, call the `GXUnlockColorProfile` function as soon as possible after manipulating the profile data.

You cannot change the size of the profile data you access with this call. If your manipulations require a change in the size of the data, you must use `GXGetColorProfile` and `GXSetColorProfile`.

This function is rarely needed. In most situations you do not need to alter the profile data of a color profile, and when you do you can use the functions `GXGetColorProfile` and `GXSetColorProfile` to make the needed changes.

ERRORS, WARNINGS, AND NOTICES

Errors

out_of_memory
colorProfile_is_nil

SEE ALSO

The `GXLockColorProfile` and `GXUnlockColorProfile` functions are described in the previous two sections.

The format of the profile data in a color profile object is described in the section “Profile Data” beginning on page 4-36. The ColorSync Utilities, including the `CMProfile` data type, are described in *Inside Macintosh: Advanced Color Imaging*.

To edit a copy of a color profile object’s profile data, rather than directly changing the data in QuickDraw GX memory, use the `GXGetColorProfile` function, described on page 4-88; to assign the edited data back to the profile, use the `GXSetColorProfile` function, described on page 4-89.

Summary of Colors and Color-Related Objects

Constants and Data Types

Color-Component Values

```
typedef unsigned short gxColorValue;
```

Color Values

```
struct gxCMYKColor{
    gxColorValue    cyan;
    gxColorValue    magenta;
    gxColorValue    yellow;
    gxColorValue    black;
};

struct gxRGBColor{
    gxColorValue    red;
    gxColorValue    green;
    gxColorValue    blue;
};

struct gxRGBAColor{
    gxColorValue    red;
    gxColorValue    green;
    gxColorValue    blue;
    gxColorValue    alpha;
};

struct gxHSVColor{
    gxColorValue    hue;
    gxColorValue    saturation;
    gxColorValue    value;
};

struct gxHLSColor{
    gxColorValue    hue;
    gxColorValue    lightness;
    gxColorValue    saturation;
};
```

```
struct gxXYZColor {
    gxColorValue  x;
    gxColorValue  y;
    gxColorValue  z;
};

struct gxYXYColor {
    gxColorValue  capY;
    gxColorValue  x;
    gxColorValue  y;
};

struct gxLUVColor {
    gxColorValue  l;
    gxColorValue  u;
    gxColorValue  v;
};

struct gxLABColor {
    gxColorValue  l;
    gxColorValue  a;
    gxColorValue  b;
};

struct gxYIQColor{
    gxColorValue  y;
    gxColorValue  i;
    gxColorValue  q;
};

struct gxGrayAColor{
    gxColorValue  gray;
    gxColorValue  alpha;
};

typedef long gxColorIndex;

struct gxIndexedColor{
    gxColorIndex  index;
    gxColorSet    set;
};
```

The Color Structure

```

struct gxColor{
    gxColorSpace      space;
    gxColorProfile    profile;
    union {
        struct gxCMYKColor    cmyk;
        struct gxRGBColor     rgb;
        struct gxRGBAColor    rgba;
        struct gxHSVColor     hsv;
        struct gxHLSColor     hls;
        struct gxXYZColor     xyz;
        struct gxYXYColor     yxy;
        struct gxLUVColor     luv;
        struct gxLABColor     lab;
        struct gxYIQColor     yiq;
        gxColorValue         gray;
        struct gxGrayAColor    graya;
        unsigned short        pixel16;
        unsigned long         pixel32;
        struct gxIndexedColor  indexed;
        gxColorValue         component[4];
    } element;
};

```

Color Packing

```

typedef enum {
    gxNoColorPacking    = 0x0000,    /* 16 bits/channel */
    gxAlphaSpace        = 0x0080,    /* space includes alpha channel */
    gxWord5ColorPacking = 0x0500,    /* 5 bits/channel, right-justified */
    gxLong8ColorPacking = 0x0800,    /* 8 bits/channel, right-justified */
    gxLong10ColorPacking = 0x0a00,   /* 10 bits/channel, right-justified */
    gxAlphaFirstPacking = 0x1000     /* alpha channel = 1st field in space */
} gxColorPackingTypes;

```

Color Spaces

```

enum gxColorSpaces{
    gxNoSpace          = 0,
    gxRGBSpace,
    gxCMYKSpace,
    gxHSVSpace,
    gxHLSpace,

```

Colors and Color-Related Objects

```

gxYXYSpace,
gxXYZSpace,
gxLUVSpace,
gxLABSpace,
gxYIQSpace,
gxNTSCSpace = gxYIQSpace,
gxPALSpace = gxYIQSpace,
gxGraySpace,
gxIndexedSpace,
gxRGBASpace = gxRGBSpace + gxAlphaSpace,
gxGrayASpace = gxGraySpace + gxAlphaSpace,
gxRGB16Space = gxWord5ColorPacking + gxRGBSpace,
gxRGB32Space = gxLong8ColorPacking + gxRGBSpace,
gxARGB32Space = gxLong8ColorPacking + gxAlphaFirstPacking
                + gxRGBASpace,
gxCMYK32Space = gxLong8ColorPacking + gxCMYKSpace,
gxHSV32Space = gxLong10ColorPacking + gxHSVSpace,
gxHLS32Space = gxLong10ColorPacking + gxHLSSpace,
gxYXY32Space = gxLong10ColorPacking + gxYXYSpace,
gxXYZ32Space = gxLong10ColorPacking + gxXYZSpace,
gxLUV32Space = gxLong10ColorPacking + gxLUVSpace,
gxLAB32Space = gxLong10ColorPacking + gxLABSpace,
gxYIQ32Space = gxLong10ColorPacking + gxYIQSpace,
gxNTSC32Space = gxYIQ32Space,
gxPAL32Space = gxYIQ32Space,
};

```

```
typedef long gxColorSpace;
```

The Color Set Object

```
typedef struct gxPrivateColorSetRecord *gxColorSet;
```

The gxSetColor Union

```

union gxSetColor{
    gxCMYKColor    cmyk;
    gxRGBColor     rgb;
    gxRGBAColor    rgba;
    gxHSVColor     hsv;
    gxHLSColor     hls;
    gxXYZColor     xyz;
    gxYXYColor     yxy;
    gxLUVColor     luv;
}

```

Colors and Color-Related Objects

```

gxLABColor      lab;
gxYIQColor      yiq;
gxColorValue    gray;
gxGrayAColor    graya;
unsigned short  pixel16;
unsigned long    pixel32;
gxColorValue    component[4];
};

```

The Color Profile Object

```
typedef struct gxPrivateProfileRecord *gxColorProfile;
```

Color Functions

```

boolean GXCheckColor      (const gxColor *source, gxColorSpace space,
                           gxColorSet aSet, gxColorProfile profile);
Fixed GXGetColorDistance  (const gxColor *target, const gxColor *source);
gxColor *GXCombineColor   (gxColor *target, gxInk operand);
gxColor *GXConvertColor   (gxColor *target, gxColorSpace space,
                           gxColorSet aSet, gxColorProfile profile);

```

Color Set Functions

Creating and Manipulating Color Set Objects

```

gxColorSet GXGetDefaultColorSet
                (long pixelDepth);
void GXSetDefaultColorSet (gxColorSet target, long pixelDepth);
gxColorSet GXNewColorSet  (gxColorSpace space, long count,
                           const gxSetColor colors[]);
void GXDisposeColorSet   (gxColorSet target);
gxColorSet GXCopyToColorSet (gxColorSet target, gxColorSet source);
boolean GXEqualColorSet   (gxColorSet one, gxColorSet two);
gxColorSet GXCloneColorSet (gxColorSet source);

```

Manipulating Color Set Object Properties

```

long GXGetColorSetOwners (gxColorSet source);
long GXGetColorSetTags   (gxColorSet source, long tagType, long index,
                           long count, gxTag items[]);

```


Colors and Color-Related Objects

```
void GXSetColorSetTags    (gxColorSet target, long tagType, long index,
                          long oldCount, long newCount,
                          const gxTag items[]);
```

Retrieving and Replacing Colors in a Color Set

```
long GXGetColorSet       (gxColorSet source, gxColorSpace *space,
                          gxSetColor colors[]);
void GXSetColorSet      (gxColorSet target, gxColorSpace space,
                          long count, const gxSetColor colors[]);
long GXGetColorSetParts  (gxColorSet source, long index, long count,
                          gxColorSpace *space, gxSetColor data[]);
void GXSetColorSetParts  (gxColorSet target, long index, long oldCount,
                          long newCount, const gxSetColor data[]);
```

Color Profile Functions

Creating and Manipulating Color Profile Objects

```
gxColorProfile GXGetDefaultColorProfile
    (void);
gxColorProfile GXNewColorProfile
    (const gxProfileRecord *profile,
     const gxProfileResponse *responses);
void GXDisposeColorProfile (gxColorProfile target);
gxColorProfile GXCopyToColorProfile
    (gxColorProfile target, gxColorProfile source);
boolean GXEqualColorProfile (gxColorProfile one, gxColorProfile two);
gxColorProfile GXCloneColorProfile
    (gxColorProfile source);
```

Manipulating Color Profile Object Properties

```
long GXGetColorProfileOwners(gxColorProfile source);
long GXGetColorProfileTags  (gxColorProfile source, long tagType,
                              long index, long count, gxTag items[]);
void GXSetColorProfileTags  (gxColorProfile target, long tagType,
                              long index, long oldCount, long newCount,
                              const gxTag items[]);
```

Retrieving and Replacing Profile Information

```
long GXGetColorProfile      (gxColorProfile source,  
                             gxProfileRecord *profile,  
                             gxProfileResponse *responses);  
  
void GXSetColorProfile     (gxColorProfile target,  
                             const gxProfileRecord *profile,  
                             const gxProfileResponse *responses);  
  
void GXLockColorProfile    (gxColorProfile source);  
void GXUnlockColorProfile  (gxColorProfile source);  
void *GXGetColorProfileStructure  
                             (gxColorProfile source, long *length);
```

Ink Objects

Contents

About Ink Objects	5-5
Ink Properties	5-6
Color	5-7
Transfer Mode	5-8
Ink Attributes	5-9
The Default Ink Object	5-10
About Transfer Modes	5-11
Transfer Mode Types	5-11
Arithmetic Transfer Modes	5-12
Highlight Transfer Mode	5-15
Boolean Transfer Modes	5-16
Pseudo-Boolean Transfer Modes	5-18
Alpha-Channel Transfer Modes	5-20
Transfer Mode Color Space	5-25
Color Limits	5-27
Source Color Limits	5-31
Destination Color Limits	5-32
Result Color Limits	5-32
Transfer Mode Matrices	5-33
Flags	5-34
Transfer Component Flags	5-35
Transfer Mode Flags	5-35
Summary of Transfer Mode Operation	5-36
Using Ink Objects	5-38
Creating and Manipulating Ink Objects	5-38
Creating and Disposing of Ink Objects	5-38
Copying, Comparing, and Cloning Ink Objects	5-39
Loading and Unloading Ink Objects	5-40

Manipulating Ink Object Properties	5-40
Getting and Setting an Ink Object's Attributes	5-40
Manipulating an Ink Object's Owner Count	5-41
Getting and Setting an Ink Object's Tag References	5-41
Getting and Setting an Ink Object's Color	5-42
Getting and Setting an Ink Object's Transfer Mode	5-43
Working With Transfer Modes	5-44
Simple Source-to-Destination Transfers	5-44
Drawing Selected Parts of the Source	5-45
Preserving Selected Parts of the Destination	5-45
Copying or Preserving Luminance	5-46
Modifying Luminance	5-47
Isolating and Modifying Color Ranges	5-47
Masking	5-48
Partial Transparency	5-48
Anti-Aliasing	5-49
Making Color Separations	5-49
Transfer Modes and Printing	5-49
Ink Objects Reference	5-50
Constants and Data Types	5-50
The Ink Object	5-50
Ink Attributes	5-51
Color Structure	5-51
Transfer Mode Structure	5-52
Transfer Mode Flags	5-53
Transfer Component Structure	5-53
Component Modes (Transfer Mode Types)	5-55
Transfer Component Flags	5-55
Functions	5-56
Creating and Manipulating Ink Objects	5-56
GXNewInk	5-56
GXDisposeInk	5-57
GXCopyToInk	5-58
GXEqualInk	5-59
GXCloneInk	5-59
Manipulating Ink Object Properties	5-60
GXResetInk	5-60
GXGetInkAttributes	5-61
GXSetInkAttributes	5-62
GXGetShapeInkAttributes	5-62
GXSetShapeInkAttributes	5-63
GXGetInkOwners	5-64
GXGetInkTags	5-65
GXSetInkTags	5-66

CHAPTER 5

Getting and Setting an Ink's Color	5-68
GXGetInkColor	5-68
GXSetInkColor	5-69
GXGetShapeColor	5-70
GXSetShapeColor	5-71
Getting and Setting an Ink's Transfer Mode	5-72
GXGetInkTransfer	5-72
GXSetInkTransfer	5-73
GXGetShapeTransfer	5-74
GXSetShapeTransfer	5-75
Summary of Ink Objects	5-77
Constants and Data Types	5-77
Functions	5-79

This chapter describes ink objects and the functions you can use to manipulate them. Read this chapter if you create or use any kind of ink object for the QuickDraw GX shapes you create. Read this chapter also if you want to understand how QuickDraw GX uses transfer modes in drawing shapes.

Before reading this chapter, you should be familiar with the information in the chapter “Introduction to QuickDraw GX” in this book. You should also be familiar with shapes, as discussed in the chapter “Shape Objects” in this book.

Although colors are contained in ink objects, they are not discussed here. Colors are discussed in the chapter “Colors and Color-Related Objects” in this book. Other than that chapter, this chapter constitutes the complete discussion of ink objects for QuickDraw GX. Unlike for shape objects and style objects, there is no separate discussion in other books of any specific graphic or typographic uses for inks.

This chapter introduces QuickDraw GX ink objects and describes their properties. It also describes how transfer modes work in QuickDraw GX. It then shows how to use the QuickDraw GX ink-manipulation functions to

- n create and manipulate ink objects
- n manipulate ink object properties
- n get and set an ink object’s color
- n work with transfer modes

About Ink Objects

An ink object exists to provide color information about a shape. Each QuickDraw GX shape consists of a shape object, a style object, an ink object, and a transform object; the ink object associated with a shape defines the color with which the shape is drawn, as well as the transfer mode used to draw it.

QuickDraw GX identifies an individual ink object through an ink **reference**. To obtain information about an ink object, you must send its reference as a parameter to a QuickDraw GX function (except that you can determine if two references identify the same ink object simply by comparing them for equality, and you can examine a reference to see if it is `nil`).

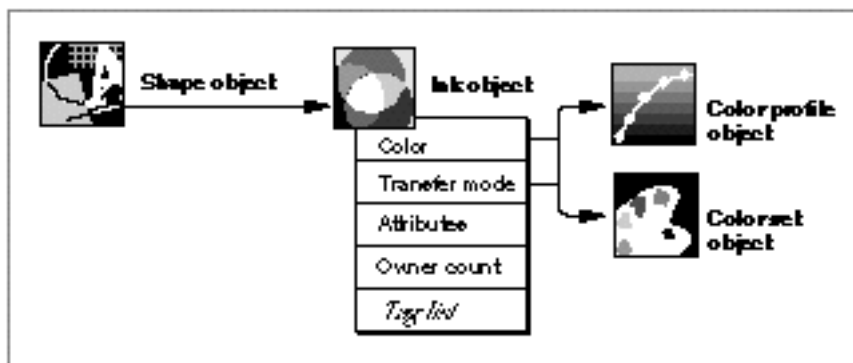
Inks are device independent. Their information is not affected by the properties of the display device to which the shapes they modify are drawn. When it draws a shape on a device, QuickDraw GX approximates as closely as possible the color specified by the shape’s ink. Device-specific color characteristics are accounted for by attaching color profiles to ink objects and by using a device’s color profile when drawing; see the chapter “Colors and Color-Related Objects” in this book for more information.

Ink Properties

The interface to ink objects is entirely procedural. You manipulate the information in an ink object by modifying its properties using QuickDraw GX functions.

Ink objects have five accessible properties, as shown in Figure 5-1. Note that, because an ink is an object and not a data structure, the order of the properties as shown in Figure 5-1 is completely arbitrary. Properties in italics are references to other objects.

Figure 5-1 The ink object and its properties



These are the five accessible properties in an ink object:

- n **Color.** A data structure that specifies the color to use for drawing the shape associated with this ink object. Besides the numeric value of the color itself, the color structure includes a specification of the color space the color is defined in terms of, as well as optional references to two other QuickDraw GX objects, a color set and a color profile.
- n **Transfer mode.** The way, or mode, of transferring the color to its destination (the screen or printed page or other location into which the shape associated with this ink is drawn). Transfer mode is a specification (such as “copy” or “XOR” or “blend”) of the interaction between the color in this ink object and the existing color or colors of the destination. With transfer mode you can make a shape opaque or transparent, draw only part of it, change its color, or combine its color with the destination color in many different ways.
The transfer mode also includes a specification of a color space and may include references to a color set and a color profile.
- n **Attributes.** A set of flags that allow you to control certain properties of view ports that affect how colors appear when a shape is drawn.
- n **Owner count.** The number of existing references to this ink object.
- n **Tag list.** A list of references to custom information about this ink object, stored in private data structures called tag objects. The chapter “Tag Objects” in this book describes tag objects in general and how you can use them to add custom information to objects.

QuickDraw GX provides functions to manipulate each of these ink object properties.

Color

One main purpose of an ink object's existence is to specify the color of a shape. Because there is only one ink object per shape, it follows that each QuickDraw GX shape can have only one color. The only exception to this is for bitmap shapes, which use pixel values rather than an ink object to specify colors. (Picture shapes have no color at all apart from the colors of their component shapes, and thus do not use their ink object.)

The color in an ink object is defined with a `gxColor` structure:

```
struct gxColor{
    gxColorSpace      space;
    gxColorProfile    profile;
    union {
        struct gxCMYKColor    cmyk;
        struct gxRGBColor     rgb;
        struct gxRGBAColor    rgba;
        struct gxHSVColor     hsv;
        struct gxHLSColor     hls;
        struct gxXYZColor     xyz;
        struct gxYXYColor     yxy;
        struct gxLUVColor     luv;
        struct gxLABColor     lab;
        struct gxYIQColor     yiq;
        gxColorValue         gray;
        struct gxGrayAColor    graya;
        unsigned short        pixel16;
        unsigned long         pixel32;
        struct gxIndexedColor  indexed;
        gxColorValue         component[4];
    } element;
};
```

The color structure specifies three characteristics of a color:

- n the color's color space, which tells what kind of format the color has—such as red-green-blue (RGB), hue-saturation-value (HSV), or luminance (grayscale).
- n a reference to a color profile object that contains information for converting the device-independent color in this ink object into color-corrected values on a particular output device. If the reference is `nil`, the QuickDraw GX default color profile is used.
- n the numeric color values that (for the given color space) specify the color of this ink object. An individual color has one number for each dimension, or color component, in the color's color space; for example, an RGB color value consists of three color component values. A color may consist of a maximum of four components.

Ink Objects

To set and manipulate the color of an ink object requires an understanding of how color works in QuickDraw GX. The color structure, color spaces, and color profiles are all described in detail in the chapter “Colors and Color-Related Objects” in this book.

Transfer Mode

The transfer mode in an ink object is contained in a `gxTransferMode` structure:

```
struct gxTransferMode{
    gxColorSpace          space;
    gxColorSet            set;
    gxColorProfile        profile;
    Fixed                 sourceMatrix[5][4];
    Fixed                 deviceMatrix[5][4];
    Fixed                 resultMatrix[5][4];
    gxTransferFlag        flags;
    struct gxTransferComponent component[4];
};
```

Like the color structure just described, the transfer mode structure specifies a color space, and may contain a reference to a color profile object or a **color set** object, which contains an array of available colors. A transfer mode specifies its own color space because it can perform its operations according to its own definitions of color, independent of the color specifications in the rest of the ink object.

The transfer mode structure contains three 5×4 matrices (5 rows, 4 columns), the **source matrix**, **device matrix**, and **result matrix**, which it can use to transform colors for special effects, by blending proportions of the colors’ components. In addition, it contains a set of **transfer mode flags** that control several aspects of the transfer mode operation.

The structure also contains up to four **transfer components**, used along with the matrices in the transfer mode operation. Transfer components contain the actual specification of the mode of transfer to use when drawing. Transfer components are defined by the `gxTransferComponent` structure:

```
struct gxTransferComponent{
    gxComponentMode      mode;
    gxComponentFlag      flags;
    gxColorValue          sourceMinimum;
    gxColorValue          sourceMaximum;
    gxColorValue          deviceMinimum;
    gxColorValue          deviceMaximum;
    gxColorValue          clampMinimum;
    gxColorValue          clampMaximum;
    gxColorValue          operand;
};
```

Ink Objects

A transfer component contains a **component mode** specifying the type of transfer mode (like “copy” or “XOR”) to use, an operand to apply (if the type calls for an operand), a set of maximum and minimum color values, and a set of flags. There is one transfer component for each color component (dimension) in the transfer mode’s color space. Each of the transfer components in the transfer mode structure may specify a different component mode, which means that each dimension of a color space can be drawn with a different transfer mode when a shape is drawn.

How these parts of the transfer mode structure and transfer component structure define the transfer mode for drawing, and how you can use transfer modes to obtain the proper effect when drawing, are described in the section “About Transfer Modes” beginning on page 5-11.

Ink Attributes

Each ink object has a set of ink attributes, a group of flags that affect the dithering and halftoning behavior when the shape associated with the ink is drawn. **Dithering** is the use of repeating patterns of differently colored pixels to simulate colors not available in a view device’s color space. **Halftoning** is the process of representing varying color intensity with evenly spaced dots of one color (but of different sizes) separated by a background of another color. The dither level and the halftone characteristics for all drawing to a view port are specified in the view port object, but you can use an ink object’s attributes to affect them for individual shapes that use that ink.

Ink attributes allow you to turn halftoning or dithering on or off, and to affect both the number of colors used in dithering and the alignment of the patterns of dithered pixels. Table 5-1 lists the ink attribute constants and describes what each one means. The constants are defined in the `gxInkAttributes` enumeration.

Table 5-1 Ink attributes

Constant	Value	Explanation
<code>gxPortAlignDitherInk</code>	<code>0x0001</code>	If set, QuickDraw GX aligns the dither pattern to the view device coordinates. If this attribute is clear (the default), QuickDraw GX aligns the dither pattern to the view port coordinates.
<code>gxForceDitherInk</code>	<code>0x0002</code>	If set, QuickDraw GX forces the dithering operation to use exactly the number of colors specified by the view port’s dither level. If this attribute is clear, QuickDraw GX may use fewer colors (in a simpler dither pattern) when constructing the dither pattern.

continued

Ink Objects

Table 5-1 Ink attributes (continued)

Constant	Value	Explanation
<code>gxSuppressDitherInk</code>	<code>0x0004</code>	If set, QuickDraw GX ignores the view port dither level, if any, and draws without dithering.
<code>gxSuppressHalftoneInk</code>	<code>0x0008</code>	If set, QuickDraw GX ignores the view port halftone, if any, and draws without creating a halftone.

IMPORTANT

Make sure that the `gxPortAlignDitherInk` attribute is cleared in ports associated with windows, so that if the window is dragged, updates using dithered drawing will match the existing parts of the drawing. (The attribute is clear by default.) s

Dithering, dither level, and halftones are described in more detail in the chapter “View-Related Devices” in this book.

The Default Ink Object

When QuickDraw GX first creates an ink object, that object has default characteristics defined by QuickDraw GX. A default ink object has the following properties:

- n No attributes set.
- n An empty tag list.
- n An owner count of 1.
- n Color space set to `gxRGBSpace` with each color component set to 0, which represents black in this color space.
- n Transfer mode set to `gxCopyMode`, with identity transfer mode matrices, color limits of 0 to 0xFFFF, and all flags cleared. Copy mode is the default transfer mode assigned to all color components of an ink object, because it is most common and fastest.

Transfer modes, matrices, color limits, and flags are described in subsequent sections of this chapter. Color spaces and color components are described in the chapter “Colors and Color-Related Objects” in this book.

To reset an ink object to its default properties, use the `GXResetInk` function, described on page 5-60.

About Transfer Modes

Basically, ink objects exist to specify two important characteristics of a shape: its color and the transfer mode to draw it with. Colors are described in the chapter “Colors and Color-Related Objects” in this book. Transfer modes are described here.

Transfer modes specify how a shape’s color is transferred onto a device. The color of a shape to be drawn (the **source color**) interacts with the existing color (the **destination color**) on the device it is drawn to. The color that results from that interaction is called the **result color**. The result color is the color of the destination after the drawing occurs.

Note that colors from different color spaces can be used. The source and destination colors are converted to the transfer mode’s color space, and the resulting color is then reconverted to the destination color space.

Bitmaps and the ink object

A bitmap shape does not use the color in its ink object, but it does use the ink’s transfer mode. Transfer modes work the same for the pixels of bitmaps as they do for colors in ink objects. For more information, see the bitmap shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*. [u](#)

QuickDraw GX allows you to influence the transfer mode operation in very flexible and powerful ways. By manipulating different parts of the transfer mode structure, you can specify

- n the type of transfer mode to apply to each color component
- n the color space in which to perform the transfer-mode calculations
- n limits on the values of color components that can be permitted in the source, destination, or result colors
- n values for the source, destination, or result matrices that can allow you to perform sophisticated transformations within and across color components
- n values for flags that affect several aspects of the transfer mode operation

The rest of this section discusses these five aspects of transfer modes. The section concludes with a summary diagram (Figure 5-18 on page 5-37) of the transfer mode process.

Transfer Mode Types

Transfer modes can be specified by type, also called **component mode**. Transfer mode types in QuickDraw GX are called component modes because QuickDraw GX allows each color component to have its own transfer mode type. In RGB color space, for example, the red component of the color may be drawn with a different transfer mode type than the blue component.

Ink Objects

QuickDraw GX supports several conceptual categories of component modes:

- n arithmetic
- n Boolean
- n pseudo-Boolean
- n highlight
- n alpha-channel

The characteristics of and most typical uses for the component modes within each category are summarized in the following subsections.

Copy mode is the default

Even though QuickDraw GX supports 18 different component modes, most applications in most situations need only one, an arithmetic mode called **copy mode**. In copy mode, the source color completely replaces the destination color. Copy mode is the default transfer mode in QuickDraw GX; therefore, you need information about other transfer modes only if you want them for special effects. u

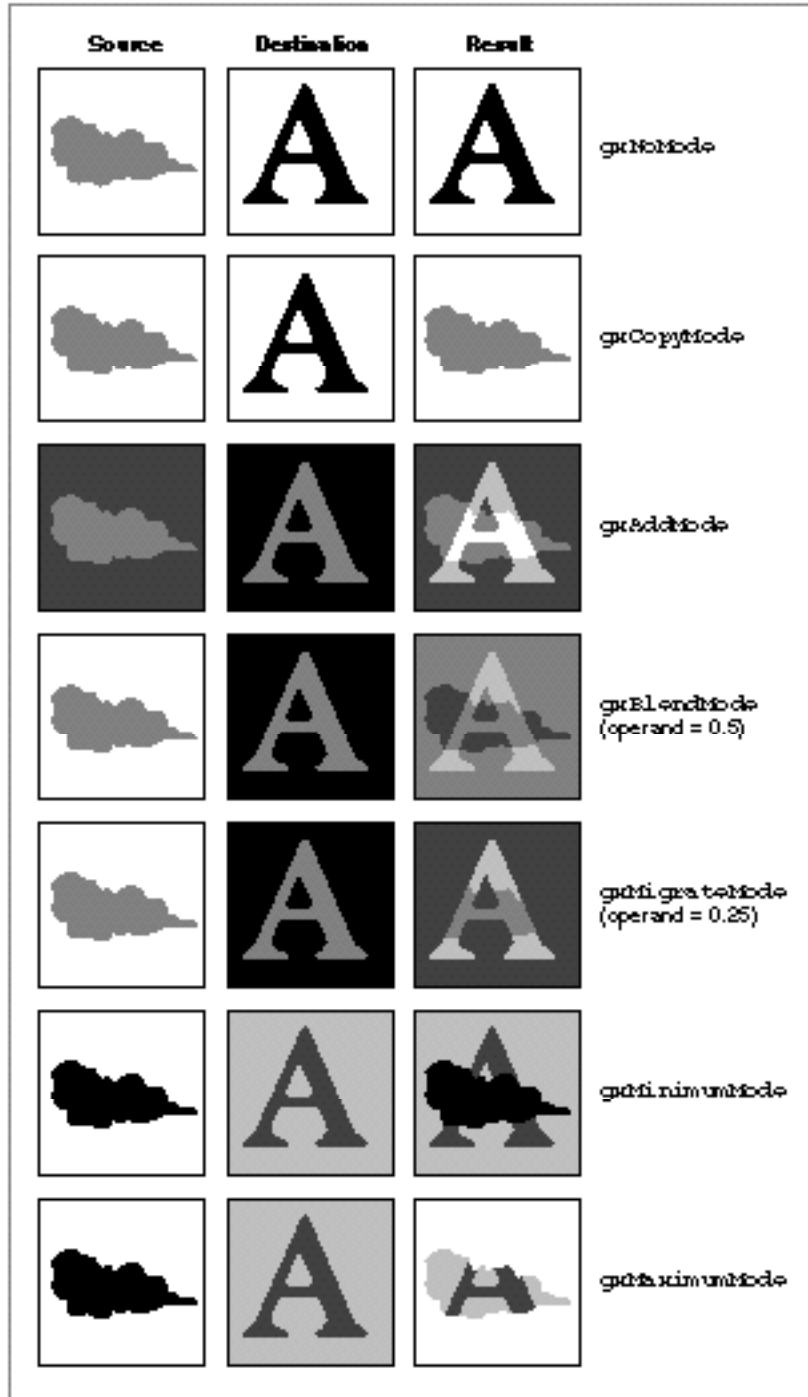
Arithmetic Transfer Modes

In arithmetic transfer modes, the numerical values of source and destination for a color component are combined arithmetically to determine the result value for that color component. In most color spaces, a color component value can vary from 0 (no intensity) to 0xFFFF (maximum intensity). You can also use the constant `gxColorValue1` to represent maximum intensity (0xFFFF).

Figure 5-2 shows examples of drawing with the arithmetic transfer modes. In each case, the source image (left) combines with the destination image (center) to produce the result image (right). You can think of the images either as two bitmaps, or as two source shapes (cloud and background) that are drawn over two destination shapes (letter and background).

Each example shows how transfer mode affects drawing within a single color component (reflected as shades of gray in the figure, where black equals 0 and white equals 0xFFFF). The constant that specifies the transfer mode type is shown to the right of each example. Note also that two of the arithmetic transfer modes use an **operand**, a numerical value that affects the outcome of the transfer-mode operation.

Figure 5-2 Arithmetic transfer modes



Ink Objects

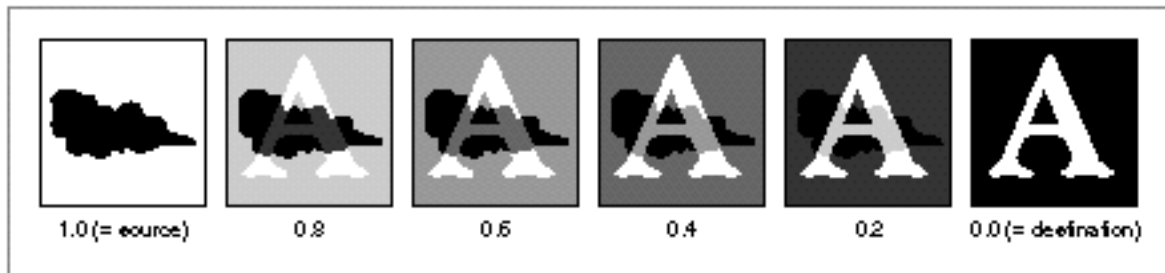
The constants that define transfer mode type are defined in the `gxComponentModes` enumeration. The arithmetic modes have the following values and meanings:

Constant	Value	Explanation
<code>gxNoMode</code>	0	No mode. No transfer occurs. For this component of the color, the destination is left as it was. This mode is useful for suppressing drawing when certain logical conditions are met, or for not drawing one color component while allowing other components to be drawn.
<code>gxCopyMode</code>	1	Copy mode. The source color component is copied to the destination. The destination component is ignored. This is the most common transfer mode, and is the default for QuickDraw GX.
<code>gxAddMode</code>	2	Add mode. The source color component is added to the destination component, but the result is not allowed to exceed the maximum value (0xFFFF or <code>gxColorValue1</code> ; white in Figure 5-2).
<code>gxBlendMode</code>	3	Blend mode. The result is the average of the source and destination color components, weighted by a ratio specified by the operand component (0.5 in Figure 5-2). The operand varies from 0 (all destination) to 0xFFFF or <code>gxColorValue1</code> (all source), although it is customary to interpret it as varying between 0 and 1.
<code>gxMigrateMode</code>	4	Migrate mode. The destination color component is moved toward the source component by the value of the step specified in the operand component (0.25, or 0x4000 in Figure 5-2). Migrate mode is similar to blend mode, except that the change in destination component is an absolute amount, rather than a proportion of the difference between it and the source component. If the source has a greater color component value than the destination, the migration is positive; if the destination has a greater value than the source, the migration is negative. In either case, the amount of migration cannot be greater than the difference between the destination and the source values.
<code>gxMinimumMode</code>	5	Minimum mode. The source component replaces the destination component only if the source component has a smaller value. (In Figure 5-2, drawing occurs only within the area occupied by the cloud.)
<code>gxMaximumMode</code>	6	Maximum mode. The source component replaces the destination component only if the source component has a larger value. (In Figure 5-2, drawing occurs only outside of the area occupied by the cloud.)

Ink Objects

The `operand` parameter is used by blend mode to specify the ratio of source and destination component. It is used by migrate mode to specify the step size by which the destination component moves toward the source component. Figure 5-3 shows examples of the result of drawing with blend mode, using several different values for the operand. (Color Plate 1 at the front of this book shows the same example in color.)

Figure 5-3 Blend example with different operand values



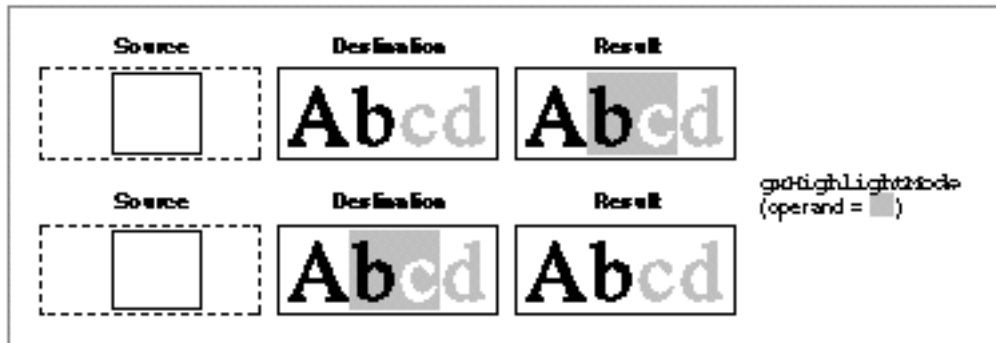
Highlight Transfer Mode

The highlight transfer mode is used for highlighting in color applications. It is most commonly used to draw (and clear) a colored rectangle around a selection, without altering the color of the item or items selected. In text, it gives the effect of drawing over the letters with a highlighting pen.

Like some of the arithmetic transfer modes, highlight mode uses an operand to control the outcome of the highlighting operation. Highlight mode operates by replacing the source color with the operand color, and the operand color with the source color, in the destination.

The upper row of images in Figure 5-4 shows a simple example of the application of highlight mode. The operand value is represented with shading rather than as a number, to illustrate how its color affects colors in the image. The source shape is a white rectangle that is drawn over the two middle letters in the destination image. (The gray letters in the line of text in the destination image represent the same color-component value as the operand, and the white area around the letters in the destination represents the same color-component value as the source.)

Figure 5-4 Highlight transfer mode



Note that black in the destination is unaffected, whereas white becomes gray and gray becomes white. A single constant specifies highlight mode, with the following value and meaning:

Constant	Value	Explanation
<code>gxHighlightMode</code>	7	Highlight mode. The source component and operand component are swapped in the destination. Other components in the destination are ignored.

In highlight mode, the source color can be thought of as the “background” color that is to be highlighted, and the operand color is the color of the highlighting pen. As the lower set of images in Figure 5-4 shows, redrawing a highlighted selection causes the source and operand colors to swap once more, effectively removing the highlighting.

The operand for highlight mode is a normal color component value that varies from 0 (no intensity) to the maximum intensity permitted for that component (normally `0xFFFF`, or `gxColorValue1`).

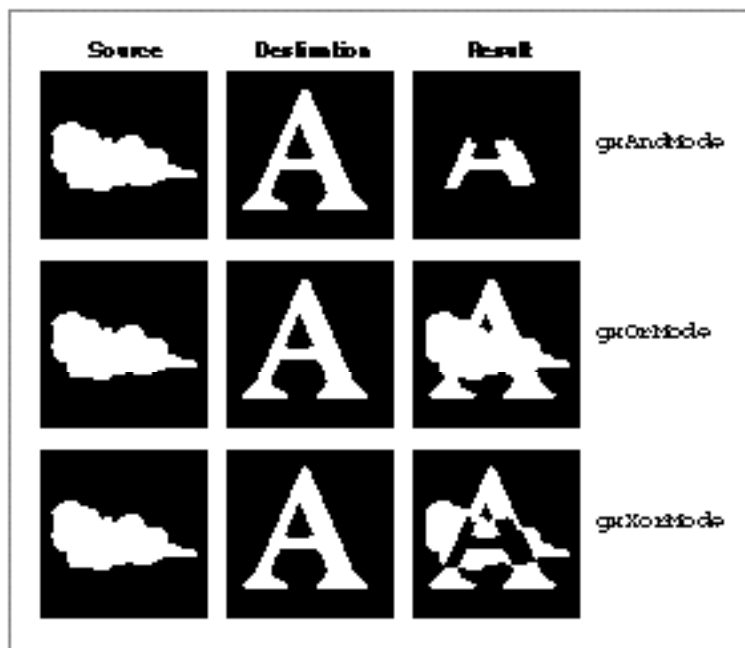
QuickDraw GX applies highlight mode only if all components in the color space specify it. An error occurs if some components specify highlight mode and others do not.

Boolean Transfer Modes

In Boolean transfer modes, the result value for a color component is determined by bit operations performed on the source and destination component values. Boolean transfer modes are most common in black-and-white drawing; in any bit depth other than 1, they yield results that can be difficult to predict because they depend on the states of the individual bits in each color-component value.

Figure 5-5 shows examples of drawing with the Boolean transfer modes at a bit depth of 1. In each case, the source image combines with the destination image to produce the result image. In these examples, black represents a bit value of 0 (clear), and white represents a bit value of 1 (set). The constant that specifies the transfer mode type is shown to the right of each example.

Figure 5-5 Boolean transfer modes (1-bit depth)



The Boolean modes have the following values and meanings:

Constant	Value	Explanation
<code>gxAndMode</code>	8	AND mode. The bits of the source color and destination color are combined using an AND operation. Only bits that are set in both source and destination remain set in the result.
<code>gxOrMode</code>	9	OR mode. The bits of the source color and destination color are combined using an OR operation. Bits that are set in either the source or the destination or in both are set in the result.
<code>gxXorMode</code>	10	XOR mode. The bits of the source color and destination color are combined using an exclusive-OR (XOR) operation. Bits that are set in the source but not the destination, and bits that are set in the destination but not the source, are set in the result. All other bits are cleared in the result.

Ink Objects

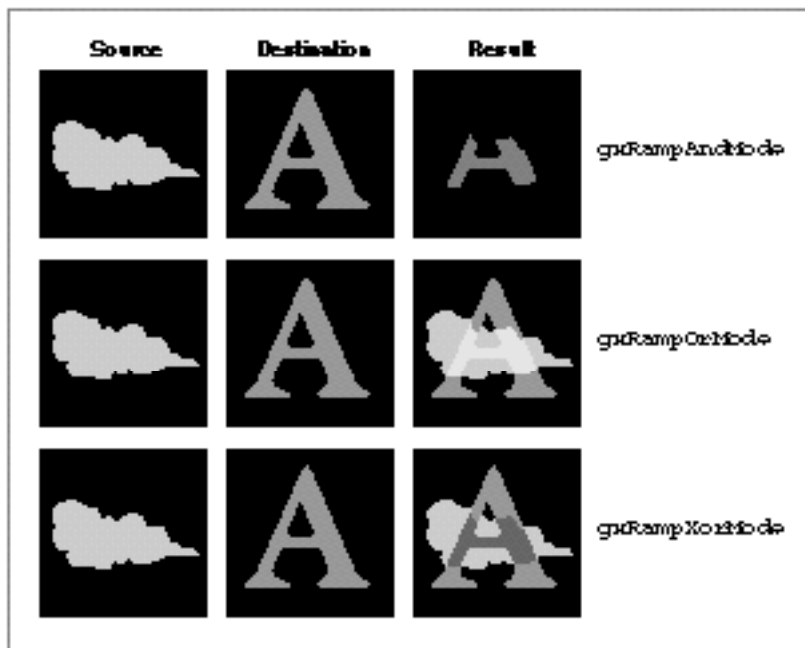
Even though they are most easily explained in terms of single-bit depths, Boolean modes are not restricted to 1-bit drawing. They can be used with any kind of color values, and are useful for manipulating colors in an indexed color space.

Pseudo-Boolean Transfer Modes

In pseudo-Boolean transfer modes, the result value for a color component is determined by normalizing the source and destination values and performing a simple arithmetic operation, to achieve consistent and predictable results analogous to 1-bit Boolean operations.

Figure 5-6 shows examples of drawing with the pseudo-Boolean transfer modes. In each case, the source image combines with the destination image to produce the result image. The constant that specifies the transfer mode type is shown to the right of each example.

Figure 5-6 Pseudo-Boolean transfer modes



Ink Objects

The constants for the pseudo-Boolean component modes have the following values and meanings:

Constant	Value	Explanation
<code>gxRampAndMode</code>	11	Ramp-AND mode. The source and destination color components are treated as ranging from 0 to 1; their product ($\text{source} \times \text{destination}$) is returned.
<code>gxRampOrMode</code>	12	Ramp-OR mode. The source and destination color components are treated as ranging from 0 to 1; the result of $(\text{source} + \text{destination} - \text{source} \times \text{destination})$ is returned.
<code>gxRampXorMode</code>	13	Ramp-XOR mode. The source and destination color components are treated as ranging from 0 to 1; the result of $(\text{source} + \text{destination} - 2 \times \text{source} \times \text{destination})$ is returned.

Note that the pseudo-Boolean and Boolean modes are similar in several ways:

- n The mode `gxRampAndMode` is similar to `gxAndMode` in that nonzero values occur in the result only where both source and destination are nonzero.
- n The mode `gxRampOrMode` is similar to `gxOrMode` in that nonzero values occur in the result wherever either the source or the destination is nonzero.
- n The mode `gxRampXorMode` is similar to `gxXorMode` in that the result is close to zero wherever the source and destination are close to each other in value.

The difference between the pseudo-Boolean and Boolean modes is that, for multi-bit pixel depths, the results for `gxRampAndMode`, `gxRampOrMode`, and `gxRampXorMode` are predictable and vary smoothly and continuously with component intensity. For 1-bit depths, these modes are identical to their Boolean equivalents.

The pseudo-Boolean modes are commonly used as component modes for alpha channels in color spaces that have an alpha channel. See “Alpha-Channel Transfer Modes” (next).

Alpha-Channel Transfer Modes

Several QuickDraw GX color spaces (`gxRGBASpace`, `gxARGB32Space` and `gxGrayASpace`) have an **alpha channel**. This is an additional color component that controls the opacity or transparency of a color. For example, a red pixel in a source image can be completely opaque, in which case it typically retains its red color when drawn over a blue pixel in the destination image. Or, the pixel can be completely transparent, in which case it typically loses all its color and turns totally blue when drawn over a blue pixel. Or, it can have an opacity of, say, `0x7FFF` (50%), in which case it typically turns magenta when drawn over a blue pixel.

Alpha channel values can be used to allow parts of one image to show through “holes” in another, to show translucency in objects that are drawn over other objects, and to perform anti-aliasing (smoothing of jagged edges) by giving feathered, semi-transparent borders to opaque objects.

When assigning transfer modes to colors with an alpha channel, you typically use two different kinds of modes:

- n To get the proper result color for each color component, you use an alpha-channel transfer mode. These modes take alpha-channel values into account when calculating result values for the color components.
- n To get the proper result opacity for the alpha channel itself, you typically use an arithmetic or pseudo-Boolean transfer mode.

This section describes how the different modes within each category work to give you the results you want.

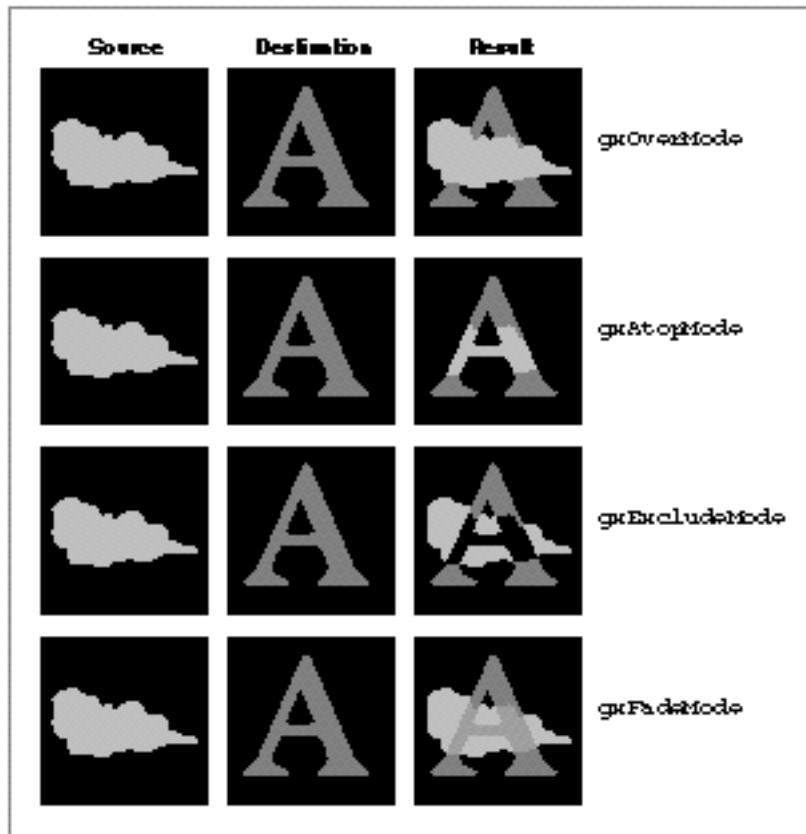
Modes for the Color Components

Figure 5-7 shows examples of how values for a color component might be calculated, given a source image and a destination image consisting of objects (or pixels) that differ in opacity. In each example, the source image (an opaque, light gray cloud against a transparent black background) combines with the destination image (an opaque, dark gray “A” on a transparent black background), to form the result image. The constant that specifies the transfer mode type is shown to the right of each example.

Ink Objects

Each example shows how the alpha-channel transfer mode affects drawing within a single color component. The mode takes into account not only the source and destination color components, but the source and destination opacities as well.

Figure 5-7 Alpha-channel transfer modes



Ink Objects

The constants for the alpha-channel component modes have the following values and meanings:

Constant	Value	Explanation
<code>gxOverMode</code>	14	Over mode. The source color is copied to the destination, and the source transparency controls where the destination color shows through. Where both are transparent, no drawing occurs (result equals destination).
<code>gxAtopMode</code>	15	Atop mode. The source color is placed over the destination, but the resulting destination retains the original destination's transparency. The effect is that opaque parts of the source are clipped to cover only opaque parts of the destination.
<code>gxExcludeMode</code>	16	Exclude mode. The destination color remains visible only where the source is transparent, and the source color is copied anywhere the destination is transparent. Where both are transparent, no drawing occurs (result equals destination); where both are opaque, the result color is 0 (no intensity).
<code>gxFadeMode</code>	17	Fade mode. The source is blended with the destination, using the relative alpha values as the ratio for the blend. Where both are transparent, the result is the average of the source and the destination).

As Figure 5-7 shows, the `gxOverMode` mode is similar to the arithmetic transfer mode `gxCopyMode`, except that it allows for transparency in the source image. Likewise, the `gxAtopMode` mode is similar to `gxCopyMode`, but it preserves the transparency of the destination image by clipping the opaque source to the destination image. The `gxExcludeMode` mode is somewhat like the Boolean transfer mode `gxXorMode`, in that opaque parts of each image appear only where the other is not opaque. The `gxFadeMode` mode is like the arithmetic `gxBlendMode`, except that the operand that controls the blend ratio is determined by the relative opacities.

Note that the images shown in Figure 5-7 are very simple and their opacities are either 0 (completely transparent) or `gxColorValue1` (completely opaque). Because an alpha component can have a wide range of partial opacities, very sophisticated translucency and color-blending effects are possible, as well as the simple masking effects shown here.

The exact formulas for determining result color are the following. In these formulas, `sA` = source alpha-channel value; `dA` = destination alpha-channel value; `sC` = source color-component value; `dC` = destination color-component value; and `rC` = result color-component value.

n For `gxOverMode`:

$$rC = (sA \times (sC - dA \times dC) + dA \times dC) / (sA + dA - sA \times dA)$$

n For `gxAtopMode`:

$$rC = dC + sA \times (sC - dC)$$

Ink Objects

n For `gxExcludeMode`:

$$rC = (sA \times sC + dA \times dC - sA \times dA \times (sC + dC)) / (sA + dA - sA \times dA)$$

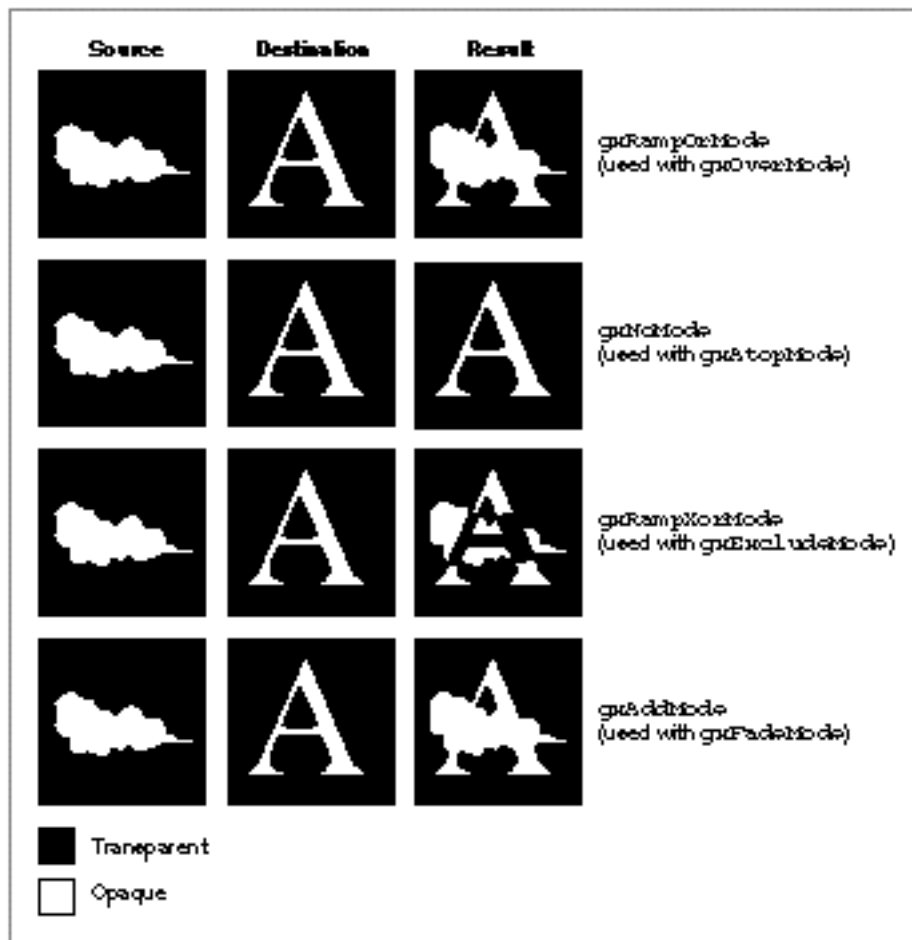
n For `gxFadeMode`:

$$rC = sA \times sC + dA \times dC / (sA + dA)$$

Modes for the Alpha Channel

For calculating the result value for the alpha channel itself, you typically use one of the arithmetic or pseudo-Boolean transfer modes presented in the previous sections—one that takes into account only the source and destination opacities. Figure 5-8 shows typical modes used and their effects on opacity, using the same images as those presented in Figure 5-7. In this figure, black represents complete transparency and white represents complete opacity. (If alpha-channel values between the two extremes existed in these examples, they would be shown in shades of gray.)

Figure 5-8 Typical modes used to determine result opacity for the alpha channel



Ink Objects

Note from Figure 5-8 that the mode you use to determine the result opacity of the alpha channel usually depends on what alpha-channel mode you use to get color-component values:

- n Use `gxRampOrMode` to calculate result alpha-channel values if you want the opacities of both source and destination summed proportionally (in a pseudo-Boolean manner; see the description of `gxRampOrMode` on page 5-19) to achieve a result opacity. Thus, if you use `gxOverMode` for the color-components, you would typically use `gxRampOrMode` for the alpha channel.
- n Use `gxNoMode` to calculate result alpha-channel values if you want the opacity of the destination to remain unchanged. Thus, if you use `gxAtopMode` for the color-components, you would typically use `gxNoMode` for the alpha channel.
- n Use `gxRampXorMode` to calculate result alpha-channel values if you want a maximum result opacity where there is a maximum difference in opacities between source and destination. Thus, if you use `gxExcludeMode` for the color-components, you would typically use `gxRampXorMode` for the alpha channel.
- n Use `gxAddMode` to calculate result alpha-channel values if you want the result opacity to reflect the sum of the opacities of the source and destination (pinned to the maximum permitted value). Thus, if you use `gxFadeMode` for the color-components, you would typically use `gxAddMode` for the alpha channel.

Note

When converting a color from a color space that does not have an alpha channel to one that does, QuickDraw GX sets the alpha channel intensity to maximum (opaque). When a color is converted from a color space that does have an alpha channel to one that does not, the alpha channel is lost. u

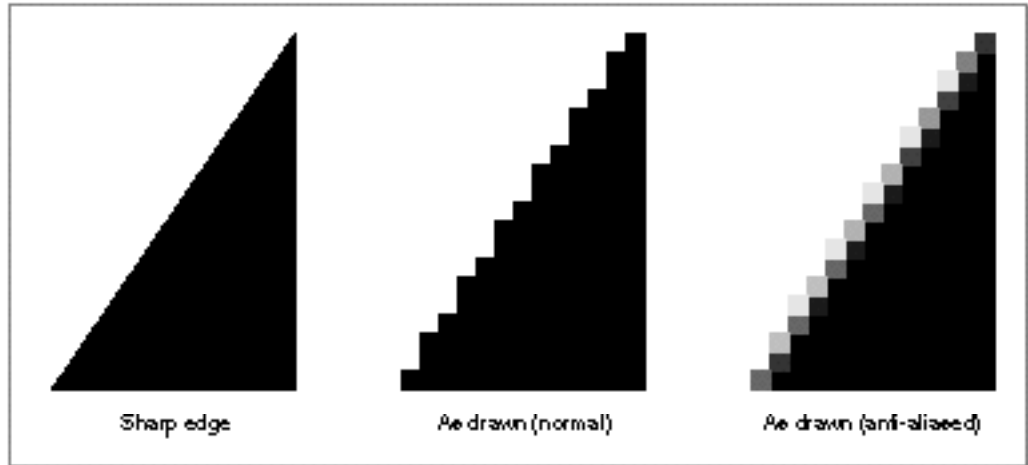
Transparency Ramps and Anti-Aliasing

Two common applications for alpha-channel colors involve making objects or images partially opaque to give a translucent effect, and smoothing jagged edges on objects drawn at low resolution.

You can create a bitmap in which the alpha-channel values of the pixels vary smoothly in one or more directions, thus creating a transparency ramp that allows the destination image to show through the source image to varying degrees across the bitmap. Color Plate 2 at the front of this book, for example, shows the kind of effect that can be achieved with a simple alpha-channel ramp.

The smoothing of jagged edges on displayed objects is called **anti-aliasing**. You can perform anti-aliasing by modifying the alpha-channel values of the pixels surrounding the edges of an opaque object. You make an individual pixel more or less opaque, based on the proportion of that pixel that the object is computed to cover.

In Figure 5-9, for example, the left image shows the computed position of the edge of a shape in a bitmap. The center image shows how that edge is displayed normally, given the resolution of the bitmap. The right image shows that edge as it might be displayed with anti-aliasing applied. The apparent jaggedness is decreased because pixels near the edge allow the background to show through to varying degrees.

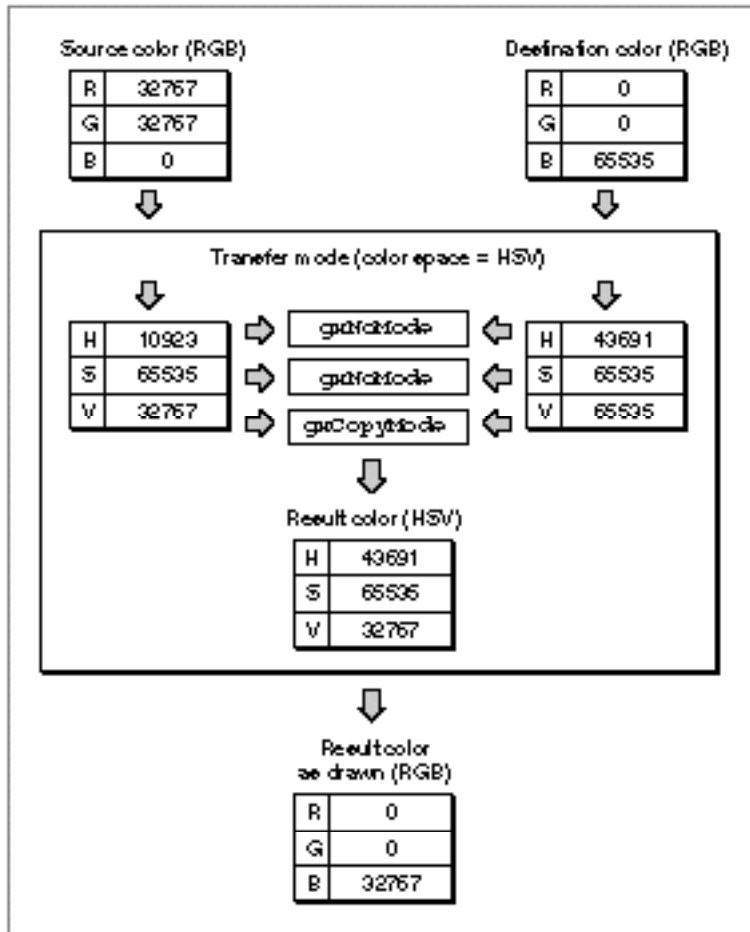
Figure 5-9 Anti-aliasing

Transfer Mode Color Space

The `space` field in the transfer mode structure specifies the color space the transfer mode calculations take place in. This space does not need to be the same as the color space specified by the ink's color, or the destination color space as specified by the view port or view device with which the ink is associated. The source and destination colors are converted into the color space that provides the context for the transfer, and the resulting color is then reconverted to the destination color space. Keep in mind that all transfer mode computations take place in the transfer mode's color space.

You needn't convert color values among different spaces yourself in order to use a different transfer mode. The transfer mode operation automatically converts colors from the ink's color space and the view device's color space, manipulates them, and then converts the result color back to the view device's color space for drawing. In creating shapes, you can work in whatever color space is convenient for you; when drawing, you can use any transfer mode color space you want; and neither color space need be the same as the color space used by the view device to which you are drawing.

Figure 5-10, for example, shows a source color in RGB space as specified in an ink object, a destination color in RGB space as specified by a monitor, and a transfer mode color space of HSV, as specified by the application. The component modes selected mean that the hue and saturation of the destination are preserved, but the value (lightness) of the source is maintained. QuickDraw GX automatically performs all necessary conversions.

Figure 5-10 Automatic conversion of color values during a transfer mode operation

The transfer mode color space defines how many components are required to perform the transfer mode operation. Monochromatic (grayscale) color spaces and indexed color space require only one component to be filled out. Alpha-channel spaces and CMYK space require four components to be filled out. All other spaces require three components to be filled out. (Color spaces are described in the chapter “Colors and Color-Related Objects” in this book.)

Remember that if the transfer mode's color space is `gxIndexedSpace`, the transfer mode structure must contain a reference to a color set object.

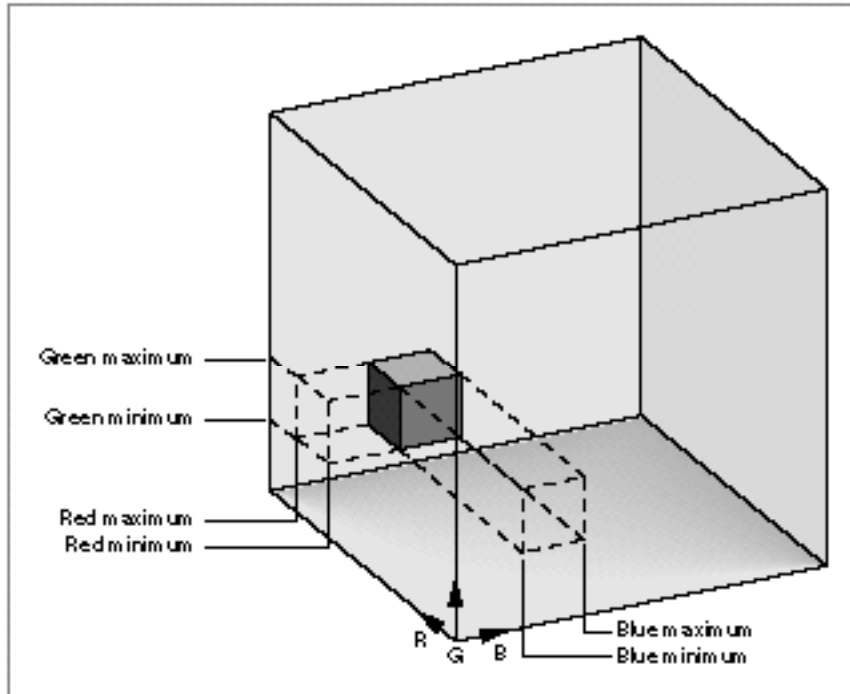
Note

Choosing a different color space can radically affect the behavior of a transfer mode. For example, if your transfer mode uses RGB space, and you have specified `gxCopyMode` for the component mode of `component[0]` and `gxNoMode` for the other components in the transfer mode structure, drawing will transfer only the red component of your source image to the destination, and leave the blue and green components of the destination as they are. If you then change the transfer mode color space to HSV and redraw, all hues in your source image will be transferred to the destination, but with the brightnesses and saturations of the original destination image. \cup

Color Limits

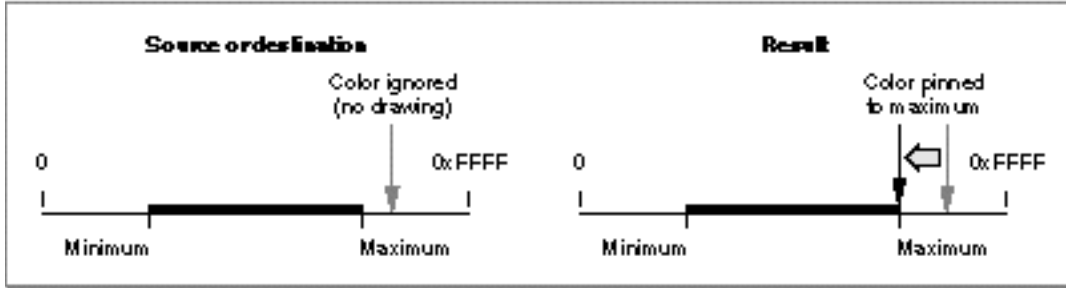
Transfer mode operations allow you to specify limits on the acceptable input values for the source or destination color, and on the acceptable output values for the result color. For example, in converting CMYK color to RGB, you may wish to limit the intensities to values that can be displayed without oversaturating the phosphors on a monitor's screen. Or, to create a special effect, you may want to draw only the extreme light and dark portions of an image, leaving out its midrange entirely.

Each color component in the `component` field of the transfer mode structure can have a maximum and a minimum permitted value. The permissible ranges can be interpreted as shown in Figure 5-11. In the figure, the large cube represents all of RGB space; the small cube represents one possible example of the limits that could be imposed on allowable values for all three components.

Figure 5-11 Maximum and minimum color-component values in RGB space

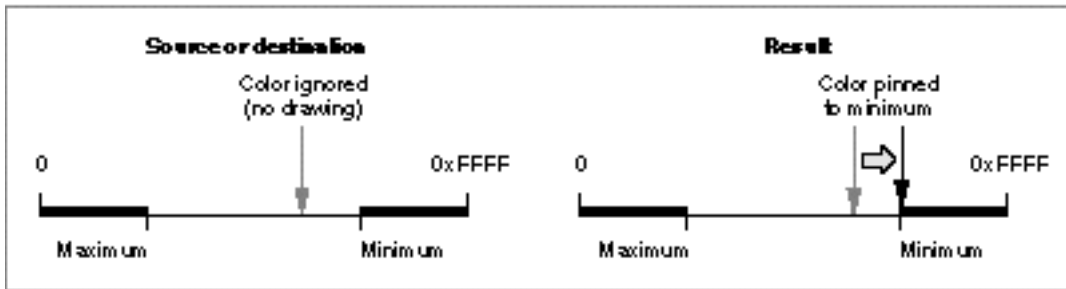
In the case of source and destination colors, color values outside the range of acceptable values (that is, outside the small cube in Figure 5-11) are ignored; if any single component value is outside of its acceptable range, no drawing occurs at all for that color. In the case of the calculated colors that result from a given transfer mode operation, color values outside of the acceptable range are **pinned** to, or moved so that they don't exceed, the nearest acceptable value (the closest edge of the small cube). See Figure 5-12.

Figure 5-12 How minimum and maximum color limits affect drawing

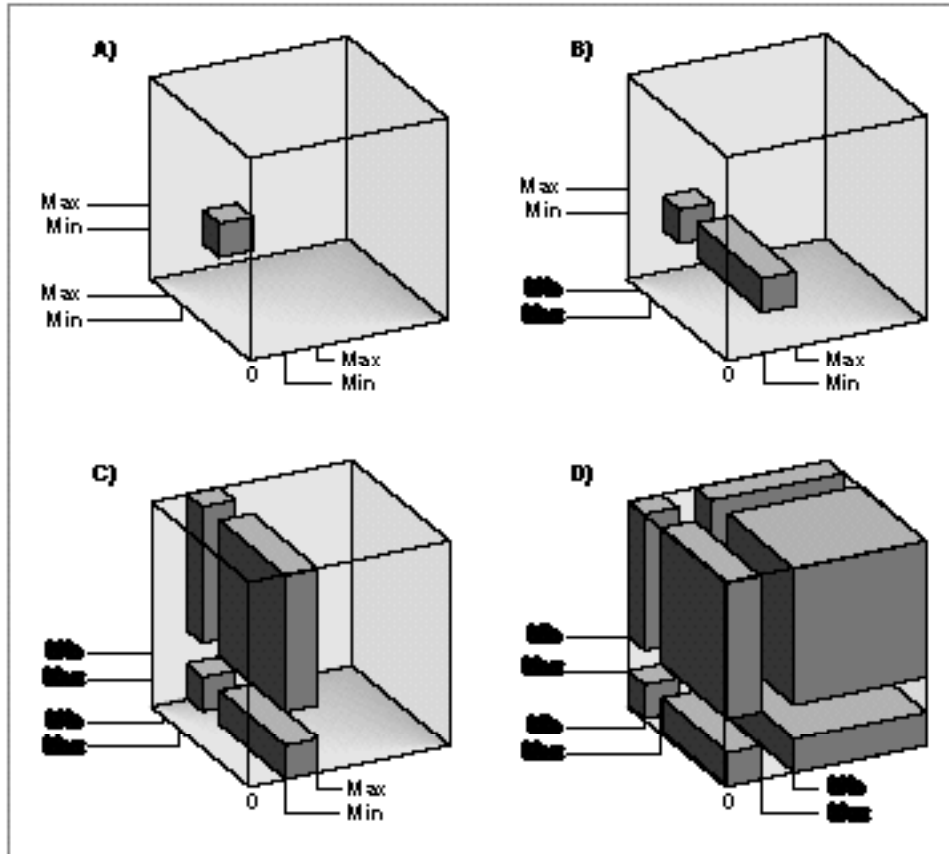


For a given component, the maximum value for a color limit can be either greater or smaller than the minimum. If the maximum is less than the minimum, only the extreme color values (that is, values *outside* of the small cube area in Figure 5-11) are allowed. See Figure 5-13.

Figure 5-13 How reversed minimum and maximum color limits affect drawing



Each of the components in a color space can have its limits set entirely independently of the others. Figure 5-14 shows the effects of reversing, in turn, the maximum and minimum values for each of the three axes in RGB space.

Figure 5-14 The effects of reversing maximum and minimum in a color space

Where the words *Min* and *Max* are bold in Figure 5-14, the minimum is greater than the maximum. Refer to Figure 5-11 on page 5-28 for the positions of the color axes on the RGB cube in this figure:

- n In drawing (A), all minimum limits are less than their respective maximums; the allowable color ranges form a small cube, just as in Figure 5-11.
- n In drawing (B), the maximum on the red axis is less than the minimum; only red color values outside of the range of the small cube are permitted, whereas blue and green must still be within the limits of the small cube. The acceptable color values form two rectangular solids within RGB space.

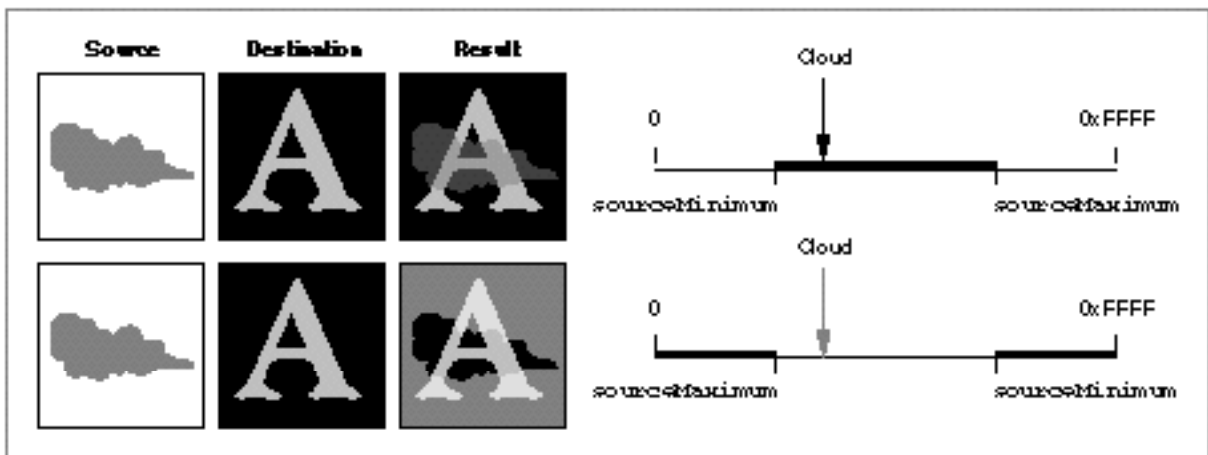
Ink Objects

- n In drawing (C), the maximum and minimum on the green axis are also reversed; the acceptable color values form a more complicated set of solids.
- n In drawing (D), all maximum and minimum color limits are reversed. In that case, only color values at the outer corners of the color space (all components outside of the range of the small cube) are acceptable.

Source Color Limits

The `sourceMinimum` and `sourceMaximum` fields in a color component's `gxTransferComponent` structure define the allowable range of values for source color in that component. Color values outside of the range cause no drawing to occur. If `sourceMaximum` is less than `sourceMinimum`, the range allowed consists of values less than `sourceMaximum` or greater than `sourceMinimum`. Figure 5-15 shows the effect of `sourceMinimum` and `sourceMaximum` on drawing using blend mode.

Figure 5-15 The effect of source color limits on drawing

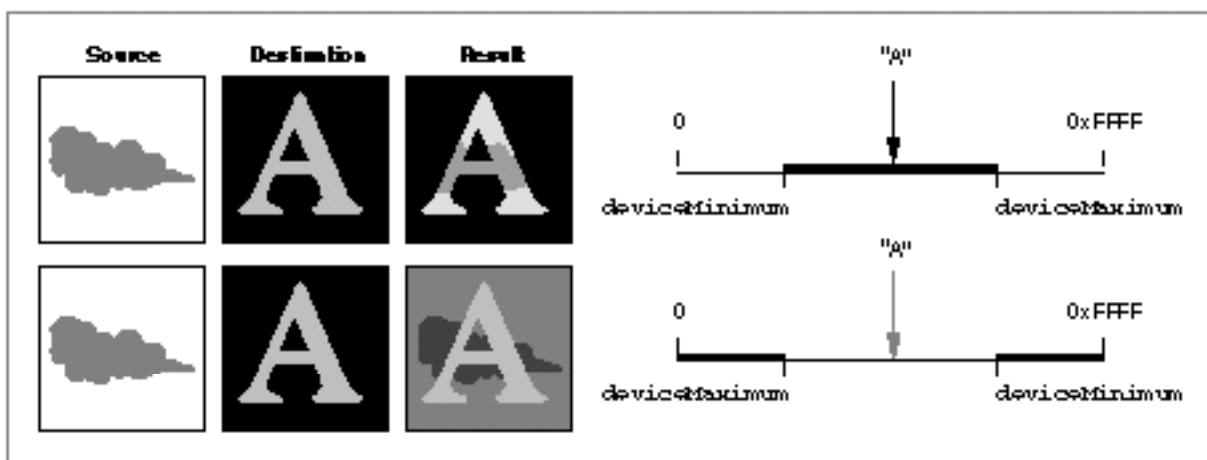


Note in Figure 5-15 that, when `sourceMinimum` is less than `sourceMaximum`, only the cloud in the source image is within the source limits, so only the cloud is blended with the destination image to create the result. Conversely, when `sourceMaximum` is less than `sourceMinimum`, the cloud in the source image is outside the source limits, so it is the only part of the source that is *not* blended with the destination image when creating the result.

Destination Color Limits

The `deviceMinimum` and `deviceMaximum` fields in a color component's `gxTransferComponent` structure define the allowable range of values for destination color in that component. Destination color values outside of the range cause no drawing to occur for that color. If `deviceMaximum` is less than `deviceMinimum`, the range allowed consists of values less than `deviceMaximum` or greater than `deviceMinimum`. Figure 5-16 shows the effect of `deviceMinimum` and `deviceMaximum` on drawing using blend mode.

Figure 5-16 The effect of destination color limits on drawing

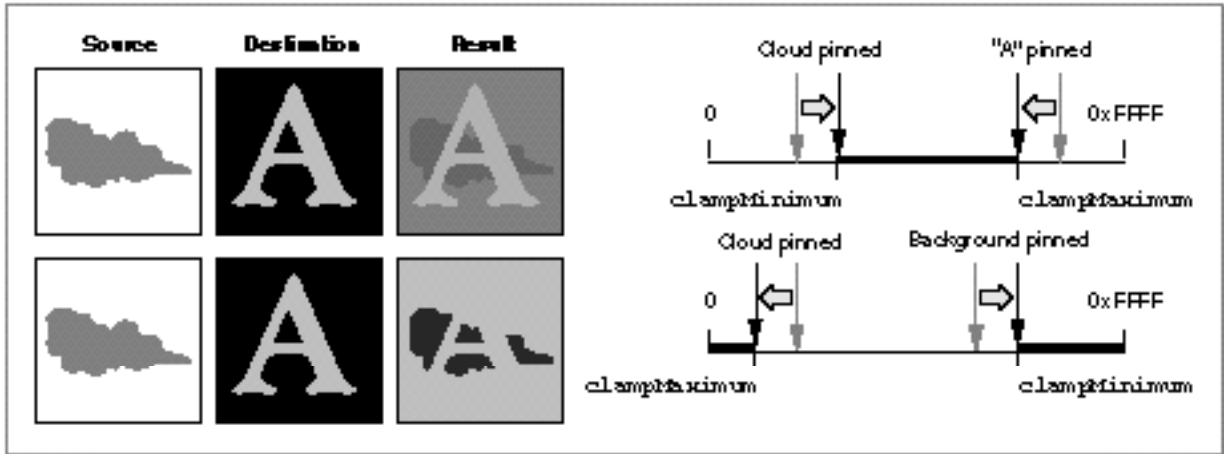


Note in Figure 5-16 that, when `deviceMinimum` is less than `deviceMaximum`, only the letter “A” in the destination image is within the destination limits, so the source is blended with the destination image only within the limits of the “A” to create the result. Conversely, when `deviceMaximum` is less than `deviceMinimum`, the “A” is outside the destination limits, so it is the only part of the destination *not* blended with the source to create the result.

Result Color Limits

The `clampMinimum` and `clampMaximum` fields in a color component's `gxTransferComponent` structure define the allowable range of values for the result color in that component. Color values outside of the range are pinned to the nearest clamp limit. If `clampMaximum` is less than `clampMinimum`, the range allowed consists of values less than `clampMaximum` or greater than `clampMinimum`. Figure 5-17 shows the effect of `clampMinimum` and `clampMaximum` on drawing using blend mode.

Figure 5-17 The effect of result color limits on drawing



Note in Figure 5-17 that, when `clampMinimum` is less than `clampMaximum`, extreme color values cannot occur in the result. The portions of the “A” outside of the cloud are darker than they would normally be with blend mode, and the portions of the cloud outside of the letter are lighter than they would normally be. Conversely, when `clampMaximum` is less than `clampMinimum`, midrange values are not possible in the result. The background in the result is lighter than it would normally be with blend mode, and the portions of the cloud outside of the “A” are darker than they would normally be.

Note

Pinning restricts the value of the computation, not necessarily the value allowed for the actual pixel. The pixel value is the closest found to the computation, which may be outside of the range of `clampMinimum` and `clampMaximum`. ^u

Transfer Mode Matrices

QuickDraw GX provides three matrices in the transfer mode structure to give you great freedom in controlling, modifying, and combining source, destination, and result color components when performing a transfer mode operation.

The **source matrix**, **device matrix**, and **result matrix** provide a way of scaling, weighting, swapping, and averaging the components of a color space before or after the transfer mode operation. Each matrix is a 5×4 array that specifies the mixture of each of the (up to 4) components, plus an offset.

Ink Objects

An **identity matrix**, one that has values of 1.0 along the diagonal and zero values elsewhere, has no effect. Here it is applied to a color in CMYK space:

$$\begin{bmatrix} c & m & y & k & 1 \end{bmatrix} \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix} = \begin{bmatrix} c & y & m & k \end{bmatrix}$$

The values for all color components after the matrix multiplication are the same as before. All transfer mode matrices in the default ink object are identity matrices.

The bottom row of the matrix specifies an offset value. The following matrix replaces c with $1/2 c + 1/2 m$; it also scales k by 0.8 and adds 0.2 to it:

$$\begin{bmatrix} c & m & y & k & 1 \end{bmatrix} \begin{bmatrix} 0.5 & 0.0 & 0.0 & 0.0 \\ 0.5 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.8 \\ 0.0 & 0.0 & 0.0 & 0.2 \end{bmatrix} = \begin{bmatrix} (0.5c + 0.5m) & m & y & (0.8k + 0.2) \end{bmatrix}$$

The source and device matrix are applied before the transfer mode calculation and after applying source minimum and source maximum. The result of the transfer mode calculation is run through the result matrix. The use of matrices allows you to apply sophisticated mapping operations—analogue to the scaling, rotation, translation, and distortion of shapes discussed in the chapter “Transform Objects” in this book—to the colors involved in a transfer mode operation. Matrices are also used to create color separations, and to map source color ranges to spot colors.

Note

Although color components are described by unsigned shorts (16-bit positive numbers), the math internal to transfer modes is performed with longs (32-bit signed numbers) to minimize overflow or roundoff error. As an example, elements in the source matrix could multiply by a large number, and elements in the result matrix could divide by a large number, without creating an overflow condition. \cup

Flags

QuickDraw GX provides two sets of flags in the transfer mode structure that control certain aspects of the transfer mode operation. One set operates on individual color components; the other set operates on the source or destination color as a whole, taking into account all of the components.

Transfer Component Flags

The transfer component flags are a set of flags in the `gxTransferComponent` structure (in the `component` field of the transfer mode structure) that alter the source, destination, or result value for an individual color component. There are two constants for these flags, defined in the `gxComponentFlags` enumeration:

Constant	Value	Explanation
<code>gxOverResultComponent</code>	<code>0x01</code>	QuickDraw GX performs an AND operation between the result color and <code>0xFFFF</code> before clamping.
<code>gxReverseComponent</code>	<code>0x02</code>	QuickDraw GX reverses the source and destination values before performing the transfer mode operation.

Specifying `gxOverResultComponent` allows the result of transfers using `gxAddMode` to wrap around (from `0xFFFF` to `0x0000`) instead of remaining clamped at `0xFFFF`.

Specifying `gxReverseComponent` allows you to apply a transfer mode backwards—from the destination to the source—for a particular component. It is most useful for component modes that depend on order, like `gxMigrateMode`, or `gxAddMode` when used for subtraction.

Transfer Mode Flags

The transfer mode flags are a set of flags in the `flags` field of the transfer mode structure. They affect how color limits are used and whether a single component mode is to be used for all color components. There are three values for the flags, defined in the `gxTransferFlags` enumeration:

Constant	Value	Explanation
<code>gxRejectSourceTransfer</code>	<code>0x0001</code>	Negate the results of <code>sourceMinimum</code> and <code>sourceMaximum</code> for all components. Accept only values outside of the specified ranges.
<code>gxRejectDeviceTransfer</code>	<code>0x0002</code>	Negate the results of <code>deviceMinimum</code> and <code>deviceMaximum</code> for all components. Accept only values outside of the specified ranges.
<code>gxSingleComponentTransfer</code>	<code>0x0004</code>	Use a single transfer component for all color components. Duplicate <code>component[0]</code> in the transfer mode structure for all components in the transfer mode's color space.

Ink Objects

Setting the `gxRejectSourceTransfer` or `gxRejectDeviceTransfer` flag causes an inversion of the acceptable color ranges for source or destination color, respectively. For example, in Figure 5-14 on page 5-30, setting the `gxRejectSourceTransfer` or `gxRejectDeviceTransfer` flag would cause the white (empty) portions of the large cubes that represent RGB space to be within range, instead of the gray (filled) portions.

The effect is similar to, although not exactly the same as, individually reversing the minimum and maximum values for the color components. If the transfer mode flag is cleared, drawing occurs only when *all* components are inside the allowed ranges—that is, inside the darker gray portions of the large cubes in Figure 5-14. With the flag set, drawing occurs any time *at least one* component is outside of its allowed range—that is, with values anywhere outside of the dark gray areas in Figure 5-14.

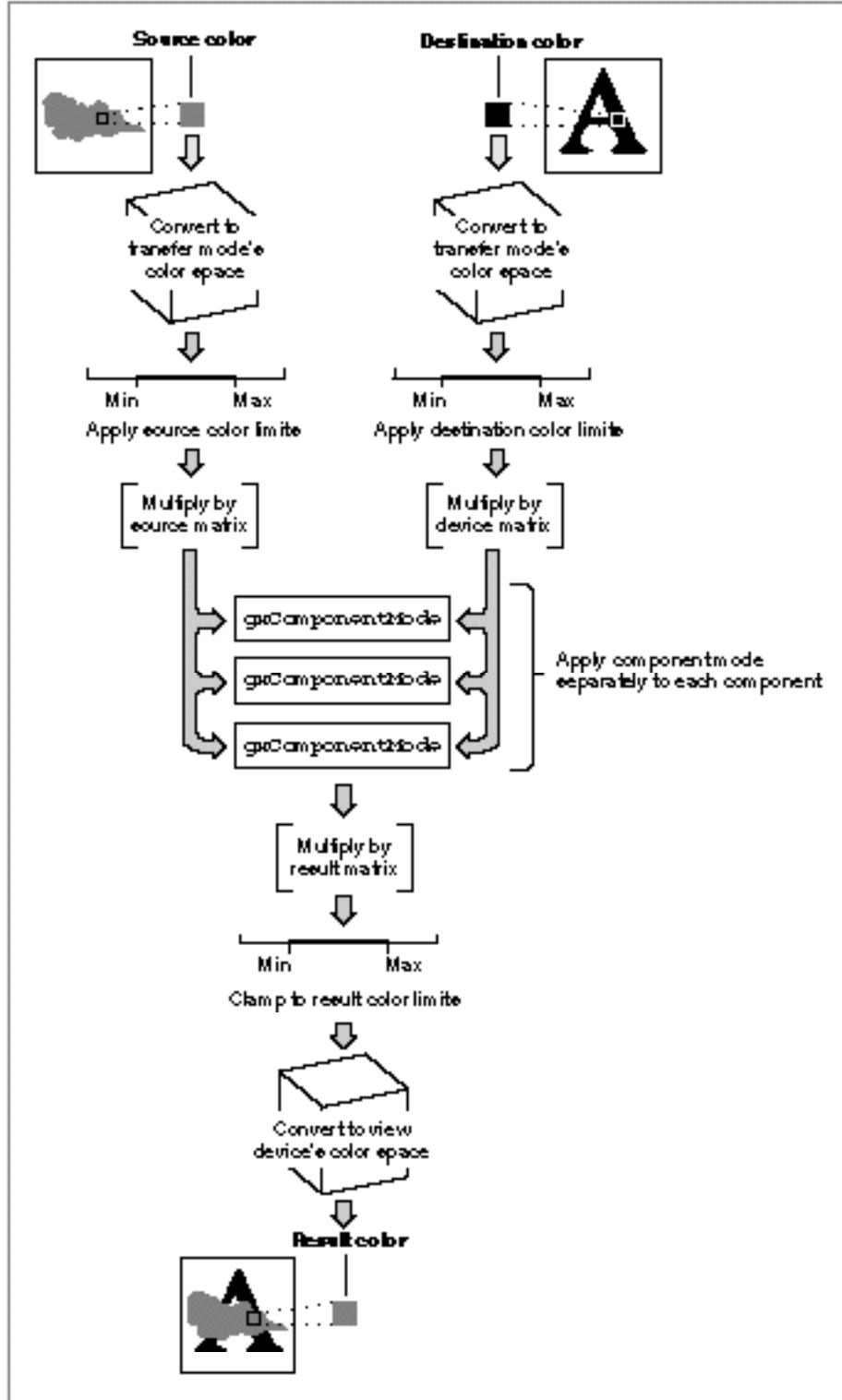
The `gxSingleComponentTransfer` flag is provided as a convenience. You can set the flag when you don't need the flexibility (and extra effort) of specifying different transfer modes for different color components. In this case you need set up only one `gxTransferComponent` structure, instead of one for each component in the transfer mode's color space.

Summary of Transfer Mode Operation

Figure 5-18 shows how all the parts of the transfer mode structure work together when a color is drawn.

1. The source color is converted to the transfer mode's color space, if they are not already the same. Each component of the source color is then compared to the acceptable range of source colors, modified if appropriate by the values of the transfer mode flags. The resulting source components are then multiplied by the source matrix to yield the corrected source components for the transfer mode operation.
2. The destination color is converted to the transfer mode's color space, if necessary. Each component of the destination color is then compared to the acceptable range of destination colors, modified if appropriate by the values of the transfer mode flags. The resulting destination components are then multiplied by the device matrix to yield the corrected destination components for the transfer mode operation.
3. The pairs of source and destination components are each combined, according to the selected component mode, and the value of the operand if the component mode takes one. (Source and destination values for a component are swapped before combining if the `gxReverseComponent` component flag is set.)
4. The components that result from the transfer mode operation are each multiplied by the results matrix to yield the corrected result components (and then ANDed with `gxColorValue1 (0xFFFF)` if the `gxOverResultComponent` flag is set for that component.) Each component is then pinned, if necessary, to the acceptable range of result colors. Finally, the result color is converted to the color space of the view device, and the color is drawn.

Figure 5-18 Summary of transfer mode operation



Using Ink Objects

This section describes how to create and use ink objects that support graphic or typographic shapes. This section describes how you can

- n create and manipulate ink objects
- n manipulate ink object properties
- n get and set an ink's color
- n work with transfer modes to achieve particular graphic effects

Creating and Manipulating Ink Objects

This section describes how you can create and interact with ink objects as whole entities—to create, dispose of, copy, compare, and clone them. Manipulating the individual properties of ink objects is described under “Manipulating Ink Object Properties” beginning on page 5-40.

Creating and Disposing of Ink Objects

QuickDraw GX provides the `GXNewInk` function to allow you to create a new ink object. You can also create a new ink object by copying an existing one with the `GXCopyToInk` function. Once you have created an ink object, you can customize its features using the techniques described in the section “Manipulating Ink Object Properties” beginning on page 5-40.

To delete your application's reference to an ink object, call the `GXDisposeInk` function, which may or may not actually release the memory allocated for that ink object, depending on the ink's owner count. The `GXDisposeInk` function decreases the ink object's owner count by 1; if that brings the owner count to zero, the ink is completely deleted and its memory released. See “Manipulating an Ink Object's Owner Count” beginning on page 5-41.

The following code fragment creates three ink objects in turn, gives each a color, assigns each to a shape (`windowShape1`, `windowShape2`, and `windowShape3`), adds each to a picture shape (`gOurHouse`), and then disposes of each ink reference because it is no longer needed. The code calls the library function `SetInkCommonColor` to assign colors to the individual ink objects.

```
gxInk redInk = GXNewInk();
SetInkCommonColor(redInk, red);
GXSetPictureParts(gOurHouse, 0, 0, 1, &windowShape1,
                 nil, &redInk, nil);
GXDisposeInk(redInk);
```


Ink Objects

```

gxInk grayInk = GXNewInk();
SetInkCommonColor(grayInk, gxGray);
GXSetPictureParts(gOurHouse, 0, 0, 1, &windowShape2,
                  nil, &grayInk, nil);
GXDisposeInk(grayInk);

gxInk turquoiseInk = GXNewInk();
SetInkCommonColor(turquoiseInk, turquoise);
GXSetPictureParts(gOurHouse, 0, 0, 1, &windowShape3,
                  nil, &turquoiseInk, nil);
GXDisposeInk(turquoiseInk);

```

The `GXNewInk` function is described on page 5-56. The `GXDisposeInk` function is described on page 5-57. The `GXSetPictureParts` function is described in the picture shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*.

Copying, Comparing, and Cloning Ink Objects

You can use the `GXCopyToInk` function to copy information from one ink object to another or to create a new copy of an ink object.

You can test if two ink-object references refer to the same ink object by simply testing the references for equality. You can also compare two different ink objects for equality with the `GXEqualInk` function. For two ink objects to be equal, their attributes, colors, color spaces, and transfer modes must have identical values, although their general object properties (owner count and tag list) do not need to be identical. Ink object copies created with the `GXCopyToInk` function are always equal to the ink from which they were copied.

The following code fragment effectively changes the default ink for a certain shape type. It makes a copy (`tempInk`) of the default ink object, modifies its color and transfer mode, and then assigns the modified copy to the default shape object for line shapes. After it makes the assignment, the code disposes of `tempInk`. The code makes use of the library functions `SetInkCommonColor` and `SetInkCommonTransfer` to set the ink's color and transfer mode.

```

tempInk = GXCopyToInk(nil,
                     GXGetShapeInk(GXGetDefaultShape(gxLineType)));
SetInkCommonColor(tempInk, gxBlack);
SetInkCommonTransfer(tempInk, xorMode);
GXSetShapeInk(GXGetDefaultShape(gxLineType), tempInk);
GXDisposeInk(tempInk);

```

In certain circumstances, you may want to copy a reference to an ink object without actually copying the ink object. For example, you may want two variables to refer to the same ink object, so that editing one of them affects both. This is called **cloning** an ink, rather than copying an ink. You can use the `GXCloneInk` function to clone an ink object.

Ink Objects

Functionally, `GXCloneInk` does nothing more than increase the owner count of an ink object. For more information about cloning objects, see the chapter “Introduction to Objects” in this book. For information on manipulating ink owner counts, see the section “Manipulating an Ink Object’s Owner Count” beginning on page 5-41 of this chapter.

The `GXCopyToInk` function is described on page 5-58. The `GXEqualInk` function is described on page 5-59. The `GXCloneInk` function is described on page 5-59.

Loading and Unloading Ink Objects

Although you rarely need to, you can influence memory-allocation decisions involving objects that you have created. If your application needs to have an ink object in memory, you can force QuickDraw GX to load it into memory. When your application no longer needs the ink object in a loaded state, you can instruct QuickDraw GX to unload it.

You call the `GXLoadInk` function to make sure that an ink object is in memory; if necessary, QuickDraw GX brings the object into memory from an unloaded state. You can call the `GXUnloadInk` function to instruct QuickDraw GX that it is free to unload the ink object at any time. These functions are described in the memory management chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

Manipulating Ink Object Properties

This section describes how to manipulate the common object properties of ink objects: attributes, owner count, and tag list.

To manipulate the color of an ink, see the section “Getting and Setting an Ink Object’s Color” beginning on page 5-42. To manipulate the transfer mode of an ink, see the section “Getting and Setting an Ink Object’s Transfer Mode” beginning on page 5-43.

For manipulating an ink object as a whole, see “Creating and Manipulating Ink Objects” beginning on page 5-38.

Getting and Setting an Ink Object’s Attributes

You can use the `GXGetInkAttributes` function to find the attributes of an existing ink object, and the `GXSetInkAttributes` function to set the attributes of an ink. The following statements return the attributes for the `source` shape’s ink:

```
gxInkAttribute whatAttributes;
whatAttributes = GXGetInkAttributes(GXGetShapeInk(source));
```

As an example, to clear the `gxSuppressHalftoneInk` attribute of an ink referenced by the variable `target`, you could use the following statement:

```
GXSetInkAttributes(target,
    GXGetInkAttributes(target) & ~gxPortAlignDitherInk);
```

Ink Objects

Conversely, to set the `gxSuppressHalftoneInk` attribute, you could use the following line of code:

```
GXSetInkAttributes(target,
    GXGetInkAttributes(target) | gxSuppressHalftoneInk);
```

Ink attributes are described in the section “Ink Attributes” beginning on page 5-9. The `GXGetInkAttributes` function is described on page 5-61. The `GXSetInkAttributes` function is described on page 5-62.

Manipulating an Ink Object’s Owner Count

The owner count of an object indicates the number of current references to that object. In general, QuickDraw GX manages owner counts for you. For example, when you create a new ink object, QuickDraw GX sets the owner count of the new ink to 1. When you assign an existing ink object to a shape, QuickDraw GX increments the ink’s owner count, corresponding to the new reference to the ink contained in the shape object.

If you want to manage an ink’s owner count directly, or if you want to know whether an ink object is shared, you can use the `GXGetInkOwners` function to determine the owner count of an ink, and the `GXCloneInk` and `GXDisposeInk` functions to change the owner count of an ink. The `GXCloneInk` function increments the ink’s owner count, and the `GXDisposeInk` function decrements the ink’s owner count, freeing the memory used by the ink if the owner count goes to 0.

The `GXGetInkOwners` function is described on page 5-64.

In the chapter “Style Objects” in this book, the section on manipulating a style object’s owner count discusses two common owner-count problems and how to avoid them. The problems are discussed in terms of style objects, but they apply equally well to ink and transform objects. Refer to that discussion if you find that ink objects you create have owner counts that are higher or lower than you expect.

Getting and Setting an Ink Object’s Tag References

You can examine the list of references to tag objects currently associated with an ink object using the `GXGetInkTags` function. Once you create a tag object, you can attach it to an ink object using the `GXSetInkTags` function. You can attach as many tag objects as you like to an ink object.

Tag objects and the basic functions for manipulating them are described in the chapter “Tag Objects” in this book. That chapter also lists the common tag types defined and reserved by Apple Computer, Inc.

The `GXGetInkTags` function is described on page 5-65. The `GXSetInkTags` function is described on page 5-66.

Getting and Setting an Ink Object's Color

You can use the `GXGetInkColor` function to retrieve the color (as a `gxColor` structure) of an existing ink object; you can use the `GXGetShapeColor` function to retrieve the color of the ink object associated with a particular shape. You can use the `GXSetInkColor` function to assign a color to an ink; you can use the `GXSetShapeColor` function to assign a color to the ink object associated with a particular shape.

To simply get the color of an existing ink (referenced here by `myInk`) requires defining a color structure, then calling the `GXGetInkColor` function to fill it:

```
gxColor myInkColor;
GXGetInkColor(myInk, &myInkColor);
```

If you want to obtain some part of the ink's color structure, such as its color space, you could make calls like this:

```
gxColorSpace myInkSpace;
gxColor myInkColor;
myInkSpace = GXGetInkColor(myInk, &myInkColor)->space;
```

Conversely, to assign some portion of an ink's color structure, such as its color profile (here called `newProfile`), you could make these calls:

```
gxColor myInkColor;
GXGetInkColor(myInk, &myInkColor);
myInkColor.profile = newProfile;
GXSetInkColor(myInk, &myInkColor);
```

You can give a shape a specific color by setting the individual values of its ink's color components. For a shape (`myShape`) whose ink uses an RGB color space, you could do something like this (with numeric color values `newRed`, `newBlue`, and `newGreen`):

```
gxColor newColor;
newColor.space = gxRGBSpace;
newColor.profile = nil;
newColor.element.rgb.red = newRed;
newColor.element.rgb.green = newGreen;
newColor.element.rgb.blue = newBlue;
GXSetShapeColor(myShape, &newColor);
```

The `GXGetInkColor` function is described on page 5-68; the `GXGetShapeColor` function is described on page 5-70. The `GXSetInkColor` function is described on page 5-69; the `GXSetShapeColor` function is described on page 5-71.

Colors and how to program with them are not described further in this chapter; for complete information, see the chapter "Colors and Color-Related Objects" in this book.

Getting and Setting an Ink Object's Transfer Mode

You can use the `GXGetInkTransfer` function to retrieve the transfer mode (as a `gxTransferMode` structure) of an existing ink object; you can use the `GXGetShapeTransfer` function to retrieve the transfer mode of the ink object associated with a particular shape. You can use the `GXSetInkTransfer` function to assign a transfer mode to an ink; you can use the `GXSetShapeTransfer` function to assign a transfer mode to the ink object associated with a particular shape.

To simply get the transfer mode of an existing ink (referenced here by `myInk`) requires defining a transfer mode structure, then calling `GXGetInkTransfer` to fill it:

```
gxTransferMode myInkTransfer;
GXGetInkTransfer(myInk, &myInkTransfer);
```

If you want to obtain some part of the ink's transfer mode structure, such as its color space, you could make calls like this:

```
gxTransferMode myInkTransfer;
gxColorSpace myTransferSpace;
myTransferSpace = GXGetInkTransfer(myInk, &myInkTransfer)->space;
```

Conversely, to set some portion of an ink's transfer mode structure, such as one of its transfer component structures (in an array here called `newComponents`), you could make these calls:

```
gxTransferMode myInkTransfer;
GXGetInkTransfer(myInk, &myInkTransfer);
myInkTransfer.component[2] = newComponents[2];
GXSetInkTransfer(myInk, &myInkTransfer);
```

You can alter a shape's transfer mode type by setting the component mode for one component of its ink's transfer mode. For example, you can change the transfer mode type of the first color component of the shape `myShape` to `gxMinimumMode` with calls like this:

```
gxTransferMode myShapeTransfer;
GXGetShapeTransfer(myShape &myShapeTransfer);
myShapeTransfer.component[0].mode = gxMinimumMode;
GXSetShapeTransfer(myShape, &myShapeTransfer);
```

The `GXGetInkTransfer` function is described on page 5-72; the `GXGetShapeTransfer` function is described on page 5-74. The `GXSetInkTransfer` function is described on page 5-73; the `GXSetShapeTransfer` function is described on page 5-75.

Transfer modes and the transfer mode structure are described in the section "About Transfer Modes" beginning on page 5-11. The next section describes how to use transfer modes to get particular drawing effects.

Working With Transfer Modes

This section describes some of the ways to use transfer modes for drawing. Note that there are many ways to use transfer modes in drawing; this section mentions only a few of the more common possibilities.

See “About Transfer Modes” beginning on page 5-11 for a discussion of the complex process by which QuickDraw GX performs transfer mode calculations.

Simple Source-to-Destination Transfers

To simply draw a color, shape, pattern, or bitmap to the destination, regardless of what the destination currently contains, use `gxCopyMode` for all color components. Use identity matrices and make sure all your source colors are within any color limits you are using. Clear all flags (except `gxSingleComponentTransfer`, if you are using it to make sure all your components use `gxCopyMode` with the same color limits and component flags). If your application is drawing or painting with a single color, `gxCopyMode` means that the color is applied without modification to all parts of the destination you are drawing to.

Copy mode is especially fast and efficient for drawing because the characteristics of the destination are not taken into account.

You can also use `gxAddMode`, `gxBlendMode`, or `gxMigrateMode` to draw the entire source, but in ways that combine the source and destination. If your application is drawing or painting with a single source color, these modes cause the drawn color or colors to be some combination of the source and the destination. For example, drawing a red apple onto a blue background, using `gxAddMode` in RGB space, results in a magenta apple against a blue background.

You can use `gxBlendMode` mode, for example, to lighten or darken all shapes in a picture by some ratio compared to the background (destination). The following code fragment sets the transfer mode of a shape (`theShape`) so that the shape’s color is blended in a given percentage (`thePercentage`, normalized to `gxColorValue1`) with the destination color.

```
gxTransferMode    shapeTransfer;
GXGetShapeTransfer(theShape, &shapeTransfer);

/* use single-component transfer, blend mode */
shapeTransfer.flags = gxSingleComponentTransfer;
shapeTransfer.component[0].mode = gxBlendMode;

shapeTransfer.component[0].flags = 0;
shapeTransfer.component[0].sourceMinimum = 0;
shapeTransfer.component[0].sourceMaximum = gxColorValue1;
shapeTransfer.component[0].deviceMinimum = 0;
shapeTransfer.component[0].deviceMaximum = gxColorValue1;
```

Ink Objects

```

shapeTransfer.component[0].clampMinimum = 0;
shapeTransfer.component[0].clampMaximum = gxColorValue1;
shapeTransfer.component[0].operand =
    ((unsigned long)gxColorValue1) * thePercentage)/100;

GXSetShapeTransfer(theShape, &shapeTransfer);

```

Drawing Selected Parts of the Source

There are many ways to transfer only parts of the source image to the destination.

You can use `gxMinimumMode` to transfer only those parts of the source that are darker than the destination; if your application is drawing or painting with a single source color, `gxMinimumMode` has the effect of darkening and coloring the lighter parts of the destination.

You can use `gxMaximumMode` to transfer only those parts of the source that are lighter than the destination; if your application is drawing or painting with a single source color, `gxMaximumMode` has the effect of lightening and coloring the darker parts of the destination.

You can use `gxCopyMode` but set source color limits so that only colors within certain ranges are transferred. If, for example, part of your source image is bright red, you can set a maximum limit on red intensity in the source; drawing will not occur where that bright red exists, and your destination image will “show through” in those places. You can work in HSV space and set limits on source luminance, so that, for example, your destination image will “show through” the highlights or the shadows of your source image after drawing.

If you are drawing in black and white, you can use `gxAndMode` or `gxRampAndMode` to transfer only those white parts of the source that are identical to the destination. Alternatively, you can use `gxXorMode` or `gxRampXorMode` to transfer only those white parts of the source that are different from the destination. The modes `gxOrMode` and `gxRampOrMode` transfer all of the white parts of the source to the destination. (For colors other than black and white, Boolean modes give unpredictable results, and pseudo-Boolean modes give results that look like blended versions of black-and-white Boolean.)

If you want to mask off parts of the source image that cannot be defined simply, in terms of color or intensity, you can use alpha-channel modes. See the section “Masking” on page 5-48.

Preserving Selected Parts of the Destination

Preserving parts of the destination image is equivalent to drawing only parts of the source, except that it is the characteristics of the destination, not the source, that determine where drawing does and does not occur.

Ink Objects

The modes `gxMinimumMode` and `gxMaximumMode` base drawing on destination characteristics as well as on source characteristics, so you can pick mode and source colors to make sure that desired parts of the destination remain unchanged after drawing. For example, if your destination has bright blue letters on a black background, you can replace that background by drawing with `gxMaximumMode` and using any color or image darker than the letters. If you are working in HSV space, you could, for example, turn the background to a different color (of any intensity) by drawing with `gxMaximumMode`, and using a color (such as yellow) whose hue value is less than that of the blue letters. You could even leave the background black and change the color of the letters by specifying `gxMinimumMode` for the saturation component and using any source color less saturated than the blue of the letters.

You can use `gxCopyMode` but set destination color limits so that drawing occurs only where the destination colors are within certain ranges. If, for example, the black parts of your destination image must be preserved, you can set a minimum limit on the luminance of the destination; drawing will not occur where that black exists, letting the black parts of the destination “show through” the source image after drawing. If you want to preserve only the summer sky in a destination image, set the destination color limits to block drawing in the blue range.

If you are drawing in black and white, you can use `gxAndMode` or `gxRampAndMode` to preserve only those white parts of the destination that are identical to the source. Alternatively, you can use `gxXorMode` or `gxRampXorMode` to preserve only those white parts of the destination that are different from the source (and to turn black any white parts of the destination that are also white in the source). The modes `gxOrMode` and `gxRampOrMode` preserve all of the white parts of the destination, adding to them all of the white parts of the source. (For colors other than black and white, Boolean modes give unpredictable results, and pseudo-Boolean modes give results that look like blended versions of black-and-white Boolean.)

Copying or Preserving Luminance

In some cases you may want to alter the colors but not the lightness of an image, or you may want to apply a color, pattern, or texture to an object in a grayscale image. What you are doing is preserving the luminance in either the source or destination, while letting other color components vary when you draw.

If, for example, your source image is a picture of a teapot, and your destination image is a color or pattern, you can color the teapot by drawing (in HSV space) with `gxNoMode` in the hue and saturation components, and `gxCopyMode` in the value (intensity) component.

Alternatively, if the color to apply is in the source image and the teapot is in the destination image, you could use `gxCopyMode` in the hue and saturation components, and `gxNoMode` in the value component. (Or you could draw as described in the previous paragraph, but set the `gxReverseComponent` flag in each component before drawing, although that is a less efficient way to draw.) Figure 5-19 shows an example of drawing by preserving destination luminance in that way. Color Plate 3 at the front of this book shows the same drawing sequence in color.

Figure 5-19 Applying color by preserving luminance in the destination

If your application is drawing or painting with a single source color, you can apply that color to the destination image—making it a single hue, but not otherwise changing the destination—by applying `gxCopyMode` to the hue component, and `gxNoMode` to the other components, including luminance.

Modifying Luminance

Modifying luminance can be used to make a dark image lighter or to make a light image darker, to increase its brightness range (contrast), or to create an overlay or blend of one image onto another, without affecting the hue or saturation of the destination. You can change the luminance of a destination image in several ways, most easily from within HSV or HLS color space.

You can draw to the destination using a source image that has any hue or saturation but the desired luminance, using `gxCopyMode` for luminance and `gxNoMode` for the other components. (That's the same as preserving the luminance of the source, as described in the previous section.)

You can draw to the destination with a source image whose luminance is equal to (or some multiple of, or some absolute amount above or below) the destination image, and use `gxAddMode` or `gxCopyMode`. The effect is to uniformly increase or decrease the luminance of the destination image. (Negative luminances are not permissible inputs to transfer mode calculations, although you can achieve the same effect by multiplying luminance by -1 in the transfer mode matrices.) The same effect can be achieved, however, by drawing with `gxNoMode` for all components and then applying the result matrix to add to the luminance or multiply it by some factor.

Isolating and Modifying Color Ranges

You can restrict drawing to individual colors or ranges of colors in a number of ways. For example, in RGB space you can draw mostly the reds by specifying very restrictive low source color limits for the other components, although this method wouldn't prevent dark non-red colors from being drawn. To draw mostly yellows, you could convert to CMYK space and do the same.

Ink Objects

In HSV space, you can set the source (or destination) color limits on the hue component, in order to draw (or preserve from drawing) an exact range of hues, independent of their brightness or saturation. You could, for instance, isolate greens in a source landscape image in order to draw fields and trees but not sky; or you could isolate flesh tones in a destination portrait image in order to change hair and clothing colors but not skin color.

You can change the colors in the range you have isolated by using the source, device, or result matrices to shift the hue component. Thus you could change green fields to shades of tan and brown, or even change tan and brown skin to shades of green.

Masking

You can draw an irregular portion of a source image over a destination image, letting the destination show around the edges of the source portion and through holes in it, in several ways. In simple situations, selecting the right transfer mode can accomplish what you need, as discussed under “Drawing Selected Parts of the Source” on page 5-45 and “Preserving Selected Parts of the Destination” on page 5-45.

Drawing parts of a complex image on a complex background usually requires the use of a mask image that controls what parts of the source get drawn and what parts do not. QuickDraw GX allows you to integrate a mask with any source or destination image by storing transparency information in the alpha channel of each pixel’s color.

You can make parts of your source image transparent and other parts opaque by placing either 0 or `gxColorValue1`, respectively, in the alpha-channel component of each pixel’s color. Then, when you draw your source image to the destination, use one of the alpha-channel transfer modes to get the masking effect you want. The mode `gxOverMode` is probably the most common transfer mode you’ll use; it’s like `gxCopyMode` with transparency added.

The destination image may have transparency also, and if you want to use the result image as a source image for subsequent drawing, you need to calculate the result alpha-channel values as well. With `gxOverMode` for the color components, the most likely mode to use on the alpha channel itself is `gxRampOrMode`.

Partial Transparency

Besides masking, you can use alpha-channel values to draw images transparently over other images, to give a translucent “stained-glass” or ghostly effect, or to draw a pattern or texture as a transparent surface effect or shadow on an image.

In simple situations, just using `gxBlendMode` or `gxMigrateMode` might give the translucent effect you need; the intersection of a white rectangle and a black rectangle is gray, which might be enough. For more sophisticated effects, you can define alpha-channel values in your source image that let the destination image show through partially or completely, and then pick an alpha-channel transfer mode that is appropriate, given the transparency of both source and destination and the transparency you want the result to retain.

Anti-Aliasing

Alpha-channel values and alpha-channel modes are also used for anti-aliasing, a drawing technique that minimizes jagged edges around objects drawn on the screen. Type at large sizes is typically drawn with anti-aliasing to smooth its appearance.

For anti-aliasing, you first set alpha-channel values to create a mask that defines the opaque parts of your image. Then, at the edges of the opaque portions, you define a zone of partial transparency, one or more pixels wide, that creates a transition from opacity within to transparency without. The color and transparency of each pixel are commonly computed based on the portion of the pixel that the opaque object is calculated to cover. Figure 5-9 on page 5-25 shows an example of this effect.

When you then draw the object with an alpha-channel transfer mode, its edges are feathered, with colors transitional between the source and destination colors. Diagonal lines and curves thus appear smoother and less jagged.

Making Color Separations

Creating color separations from images is fundamentally straightforward with QuickDraw GX. For CMYK color separations, you can use a transfer mode structure that works with CMYK color space and draw your image four times, each time using `gxCopyMode` for all components but modifying the source matrix each time to pass through only the component that you want.

You can also create halftones for each separation. See the chapter “View-Related Devices” in this book for a discussion of halftones in QuickDraw GX.

Transfer Modes and Printing

Printers are imaging devices, and the printing components of QuickDraw GX support all transfer modes. When an image is printed, the original state of the destination (paper) is treated as if it were a white image—equivalent to `0xFFFF` or `gxColorValue1` in all components of RGB space, for example. QuickDraw GX then accumulates all drawing commands before actually printing the page. As it draws each shape on the page, QuickDraw GX combines the source image with the destination, which may be white or may reflect the results of previous drawing, according to the transfer mode selected. After the last shape is drawn, QuickDraw GX prints the result. Thus all transfer modes, and even alpha-channel values, can be accounted for in printing.

Even vector devices such as plotters can support transfer modes. The color of each shape the plotter is to draw is combined with the destination according to whatever transfer mode is selected, and at the end the resulting color is printed using one of the available pen colors and patterns.

Ink Objects

Printing in QuickDraw GX is optimized for certain transfer mode configurations. In general, printing is fastest using the default ink object's transfer mode (`gxCopyMode` for all components, identity matrices, wide-open color limits). In addition, for fastest printing of 1-bit-per-pixel bitmaps, QuickDraw GX recognizes a special configuration that replicates the QuickDraw `srcOr` transfer mode on the Macintosh: `gxCopyMode` in all components and source color limits set so that only the “on” or “foreground” bits of the image are printed.

For more information on printing for applications, see *Inside Macintosh: QuickDraw GX Printing*. For more information on printing for printer drivers and extensions, see *Inside Macintosh: QuickDraw GX Printing Extensions and Drivers*.

Ink Objects Reference

This section provides reference information about the data structures and functions that allow you to create and manipulate ink objects, and to work with transfer modes. It includes

- n descriptions of the constants and data types used by ink objects
- n descriptions of the QuickDraw GX functions that operate on ink objects, including those that manipulate ink color and transfer mode

Constants and Data Types

This section describes the constants and data types that define

- n ink attributes
- n the color structure
- n transfer modes

The documentation for transfer modes is complete in this section; the documentation for colors is summary only. Additional reference information on colors is found in the chapter “Colors and Color-Related Objects” in this book.

The Ink Object

QuickDraw GX provides you with access to an individual ink object through an ink reference:

```
typedef struct gxPrivateInkRecord *gxInk;
```

In this type definition, `gxInk` is a type-checked reference, not an actual pointer to any defined structure. The contents of the ink object are private.

Ink Attributes

Each ink object has a set of ink attributes, defined in the `gxInkAttributes` enumeration. The attributes specify how to handle dithering and halftoning.

```
enum gxInkAttributes{
    gxPortAlignDitherInk = 0x0001,
    gxForceDitherInk     = 0x0002,
    gxSuppressDitherInk  = 0x0004,
    gxSuppressHalftoneInk= 0x0008
};

typedef long gxInkAttribute;
```

The individual ink attributes are described in the section “Ink Attributes” beginning on page 5-9.

Color Structure

The color property of an ink object is specified with a color structure (type `gxColor`):

```
struct gxColor{
    gxColorSpace space;
    gxColorProfile profile;
    union {
        struct gxCMYKColor    cmyk;
        struct gxRGBColor     rgb;
        struct gxRGBAColor    rgba;
        struct gxHSVColor     hsv;
        struct gxHLSColor     hls;
        struct gxCIEColor     cie;
        struct gxYIQColor     yiq;
        gxColorValue          gray;
        struct gxGrayaColor    graya;
        unsigned short        pixel16;
        unsigned long         pixel32;
        struct gxIndexedColor indexed;
        gxColorValue          component[4];
    } element;
};
```

The fields of the color structure are described briefly in the section “Color” beginning on page 5-7 of this chapter, and defined in more detail in the chapter “Colors and Color-Related Objects” in this book.

Transfer Mode Structure

The transfer mode structure (data type `gxTransferMode`) specifies the transfer mode property of an ink object.

```
struct gxTransferMode{
    gxColorSpace           space;
    gxColorSet             set;
    gxColorProfile         profile;
    Fixed                  sourceMatrix[5][4];
    Fixed                  deviceMatrix[5][4];
    Fixed                  resultMatrix[5][4];
    gxTransferFlag         flags;
    struct gxTransferComponent component[4];
};
```

Field descriptions

<code>space</code>	The color space used by this transfer mode. The color spaces defined by QuickDraw GX are listed in the color structure definition shown in the previous section, and are described in the chapter “Colors and Color-Related Objects” in this book. A transfer mode’s color space need not be the same as the color space of the ink object the transfer mode is part of.
<code>set</code>	A reference to a color set, an object that defines a list of colors available for use by this transfer mode. This field has meaning only if the transfer mode’s color space is <code>gxIndexedSpace</code> ; if the value in this field is <code>nil</code> , there is no color set associated with the transfer mode. Color sets are described in the chapter “Colors and Color-Related Objects” in this book.
<code>profile</code>	A reference to a color profile, an object that specifies color-correction information. If the value in this field is <code>nil</code> , there is no color profile associated with the color space of this transfer mode. Color profiles are described in the chapter “Colors and Color-Related Objects” in this book.
<code>sourceMatrix</code>	A 5 x 4 matrix that specifies how to transform the source color before applying the component modes to the components. See “Transfer Mode Matrices” beginning on page 5-33.
<code>deviceMatrix</code>	A 5 x 4 matrix that specifies how to transform the destination color before applying the component modes to the components. See “Transfer Mode Matrices” beginning on page 5-33.
<code>resultMatrix</code>	A 5 x 4 matrix that specifies how to transform the result color after applying the component modes to all components. See “Transfer Mode Matrices” beginning on page 5-33.
<code>flags</code>	The transfer mode flags; they specify how to handle color limits and whether multiple transfer components are needed. The transfer mode flags are described in the section “Transfer Mode Flags” on page 5-35.

Ink Objects

`component` An array of four transfer component structures, each specifying the type of transfer mode to apply to a single color component of the transfer mode's color space. The transfer component structure is described next.

Transfer Mode Flags

The transfer mode flags are defined in the `gxTransferFlags` enumeration:

```
enum gxTransferFlags{
    gxRejectSourceTransfer = 0x0001, /* 1 source component
                                       must be out of range */
    gxRejectDeviceTransfer = 0x0002, /* 1 device component
                                       must be out of range */
    gxSingleComponentTransfer= 0x0004 /* transfer component[0]
                                       = all components */
};

typedef long gxTransferFlag;
```

The individual transfer mode flags are described in the section “Transfer Mode Flags” on page 5-35.

Transfer Component Structure

There are up to four transfer component structures in a transfer mode structure. Each transfer component structure describes how the transfer mode operation is to be applied to a given component in the transfer mode's color space. The transfer component structure is of data type `gxTransferComponent`:

```
struct gxTransferComponent{
    gxComponentMode    mode;
    gxComponentFlag    flags;
    gxColorValue       sourceMinimum;
    gxColorValue       sourceMaximum;
    gxColorValue       deviceMinimum;
    gxColorValue       deviceMaximum;
    gxColorValue       clampMinimum;
    gxColorValue       clampMaximum;
    gxColorValue       operand;
};
```

Ink Objects

Field descriptions

<code>mode</code>	The component mode (type of transfer mode, such as <code>gxCopyMode</code> or <code>gxBlendMode</code>) to apply to this color component. Component modes are described in the section “Transfer Mode Types” beginning on page 5-11.
<code>flags</code>	The component flags, which control clamping behavior and whether source and destination are to be reversed for this component. The component flags are described in the section “Transfer Component Flags” on page 5-35.
<code>sourceMinimum</code>	The minimum acceptable value for source color in this color component. No drawing occurs if the source component value is below <code>sourceMinimum</code> . For more information, see “Color Limits” beginning on page 5-27, and “Source Color Limits” on page 5-31.
<code>sourceMaximum</code>	The maximum acceptable value for source color in this color component. No drawing occurs if the source component value is greater than <code>sourceMaximum</code> . For more information, see “Color Limits” beginning on page 5-27, and “Source Color Limits” on page 5-31.
<code>deviceMinimum</code>	The minimum acceptable value for destination color in this color component. No drawing occurs if the destination component value is below <code>deviceMinimum</code> . For more information, see “Color Limits” beginning on page 5-27, and “Destination Color Limits” on page 5-32.
<code>deviceMaximum</code>	The maximum acceptable value for destination color in this color component. No drawing occurs if the destination component value is greater than <code>deviceMaximum</code> . For more information, see “Color Limits” beginning on page 5-27, and “Destination Color Limits” on page 5-32.
<code>clampMinimum</code>	The minimum acceptable value for result color in this color component. If the result component value is below <code>deviceMinimum</code> , it is pinned, or clamped, to the value of <code>deviceMinimum</code> . For more information, see “Color Limits” beginning on page 5-27, and “Result Color Limits” on page 5-32.
<code>clampMaximum</code>	The maximum acceptable value for result color in this color component. If the result component value is greater than <code>deviceMaximum</code> , it is pinned, or clamped, to the value of <code>deviceMaximum</code> . For more information, see “Color Limits” beginning on page 5-27, and “Result Color Limits” on page 5-32.
<code>operand</code>	A value used as an input to the <code>gxBlendMode</code> , <code>gxMigrateMode</code> , and <code>gxHighlightMode</code> component modes. If you are using these modes, you must supply a proper value for <code>operand</code> . For more information, see “Arithmetic Transfer Modes” beginning on page 5-12, and “Highlight Transfer Mode” on page 5-15.

Component Modes (Transfer Mode Types)

QuickDraw GX supports the following types of transfer modes, defined in the `gxComponentModes` enumeration:

```
enum gxComponentModes{
    gxNoMode = 0,
    gxCopyMode,
    gxAddMode,
    gxBlendMode,
    gxMigrateMode,
    gxMinimumMode,
    gxMaximumMode,
    gxHighlightMode,
    gxAndMode,
    gxOrMode,
    gxXorMode,
    gxRampAndMode,
    gxRampOrMode,
    gxRampXorMode,
    gxOverMode,
    gxAtopMode,
    gxExcludeMode,
    gxFadeMode
};

typedef unsigned char gxComponentMode;
```

The individual component modes are described in the section “Transfer Mode Types” beginning on page 5-11.

Transfer Component Flags

The transfer component flags are part of the transfer component structure (data type `gxTransferComponent`). The flags are defined in the `gxComponentFlags` enumeration:

```
enum gxComponentFlags{
    gxOverResultComponent= 0x01, /* AND result component with
                                   0xFFFF before clamping */
    gxReverseComponent    = 0x02 /* Reverse source and device
                                   components before mode */
};

typedef unsigned char gxComponentFlag;
```

The individual transfer component flags are described in the section “Transfer Component Flags” on page 5-35.

Functions

This section describes the QuickDraw GX functions you can use to

- n create and manipulate an ink object
- n manipulate the common object properties of an ink object
- n get and set the color of an ink object
- n get and set the transfer mode of an ink object

Creating and Manipulating Ink Objects

This section describes the functions that manipulate inks as objects in memory. With the functions in this section, you can create and dispose of an ink object, and copy, compare, and clone ink objects.

To associate an ink object with a QuickDraw GX shape object, use the `GXGetShapeInk` and `GXSetShapeInk` functions, described in the chapter “Shape Objects” in this book.

GXNewInk

You can use the `GXNewInk` function to create a new ink object with default properties.

```
gxInk GXNewInk(void);
```

function result A reference to a newly created copy of the default ink object.

DESCRIPTION

The `GXNewInk` function creates an ink object with an owner count of 1. All other properties of the ink are set to their default values:

- n No attributes set.
- n An empty tag list.
- n An owner count of 1.
- n Color space set to `gxRGBSpace` with each color component set to 0, which represents black in this color space.
- n Transfer mode set to `gxCopyMode`, with identity transfer mode matrices, color limits of 0 to 0xFFFF (`gxColorValue1`), and all flags cleared.

SPECIAL CONSIDERATIONS

If no error occurs, the `GXNewInk` function creates an ink object; you are responsible for disposing of that object when you no longer need it.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`

SEE ALSO

Default ink values are described in the section “The Default Ink Object” on page 5-10.

GXDisposeInk

You can use the `GXDisposeInk` function to release a reference to an ink object.

```
void GXDisposeInk(gxInk target);
```

`target` A reference to the ink object to dispose of.

DESCRIPTION

The `GXDisposeInk` function decrements the owner count of the ink object specified by the `target` parameter and releases any memory used by it if the owner count goes to 0.

ERRORS, WARNINGS, AND NOTICES**Errors**

`ink_is_nil`

SEE ALSO

Owner counts for ink objects are discussed in the section “Copying, Comparing, and Cloning Ink Objects” beginning on page 5-39, and in the section “Manipulating an Ink Object’s Owner Count” beginning on page 5-41. To examine the owner count of an ink, use the `GXGetInkOwners` function, described on page 5-64.

GXCopyToInk

You can use the `GXCopyToInk` function to create a copy of an existing ink object.

```
gxInk GXCopyToInk(gxInk target, gxInk source);
```

`target` A reference to the ink object to copy the `source` contents into. If you specify `nil` for this parameter, the `GXCopyToInk` function creates a new ink object.

`source` A reference to the ink object whose contents you want to copy.

function result A reference to the copy (that is, the target ink).

DESCRIPTION

The `GXCopyToInk` function copies the contents of an existing ink object to another, or it creates a new ink object and copies the contents of an existing ink object to it. The function copies the color, transfer mode, attributes, and tag list (but not the owner count) of the ink object specified by the `source` parameter into the ink object specified by the `target` parameter. It clones, but does not copy, the tag objects in the tag list.

If you specify `nil` for the `target` parameter, the `GXCopyToInk` function creates a new ink object and copies the source properties, including owner count and tag list, into it.

You can use the `GXCopyToInk` function to create a copy of an ink object so you can modify it without changing the original.

SPECIAL CONSIDERATIONS

If you specify `nil` for the `target` parameter and no error occurs, the `GXCopyToInk` function creates an ink object; you are responsible for disposing of that object when you no longer need it.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`ink_is_nil`

SEE ALSO

To create a new ink that is a copy of the default ink instead of a copy of an existing ink, use the `GXNewInk` function, described on page 5-56.

To compare two ink objects, use the `GXEqualInk` function, described in the next section.

GXEqualInk

You can use the `GXEqualInk` function to determine whether two ink objects are equal.

```
boolean GXEqualInk(gxInk one, gxInk two);
```

`one` A reference to one of the ink objects to test for equality.

`two` A reference to the other ink object to test for equality.

function result `true` if the ink specified by the `one` parameter is equal to the ink specified by the `two` parameter; otherwise `false`.

DESCRIPTION

The `GXEqualInk` function returns as its function result a Boolean value indicating whether the ink object specified by the `one` parameter is equal to the ink object specified by the `two` parameter.

For two ink objects to be equal, they must have identical colors, transfer modes, and attributes, although their owner count and tag list need not be identical.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`ink_is_nil`

SEE ALSO

To make a copy of an ink object that is equal by the criteria of this function, use the `GXCopyToInk` function, described in the previous section.

GXCloneInk

You can use the `GXCloneInk` function to clone an ink object—that is, to add a reference to it and increment its owner count.

```
gxInk GXCloneInk(gxInk source);
```

`source` A reference to the ink object to clone.

function result A reference to the cloned ink.

DESCRIPTION

The `GXCloneInk` function increments the owner count of the ink referenced in the `source` parameter. You typically use this function when you want to create another reference to an existing ink instead of creating a distinct copy of the ink.

This function returns as its function result a reference to the ink—the same reference you pass in as the `source` parameter. It also increments the ink's owner count.

ERRORS, WARNINGS, AND NOTICES**Errors**

`ink_is_nil`

SEE ALSO

Owner counts for ink objects are discussed in the section “Copying, Comparing, and Cloning Ink Objects” beginning on page 5-39, and in the section “Manipulating an Ink Object's Owner Count” beginning on page 5-41.

To examine the owner count of an ink, use the `GXGetInkOwners` function, described on page 5-64. To decrement the owner count of an ink, use the `GXDisposeInk` function, described on page 5-57.

Manipulating Ink Object Properties

The functions described in this section allow you to

- n reset an ink's properties to their default values
- n manipulate the common object properties of an ink object: attributes, owner count, and tag list

GXResetInk

You can use the `GXResetInk` function to reset the properties of an ink to their default values.

```
void GXResetInk(gxInk target);
```

`target` A reference to the ink object whose properties you want to reset.

DESCRIPTION

The `GXResetInk` function resets the color, transfer mode, and attributes of the ink object specified by the `target` parameter to match the properties of the default ink object:

- n No attributes set.

Ink Objects

- n Color space set to `gxRGBSpace` with each color component set to 0, which represents black in this color space.
- n Transfer mode set to `gxCopyMode`, with identity transfer mode matrices, color limits of 0 to 0xFFFF (`gxColorValue1`), and all flags cleared.

The `GXResetInk` function does not change the target ink's owner count or tag list.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`ink_is_nil`

SEE ALSO

Default ink properties are described in the section “The Default Ink Object” on page 5-10.

GXGetInkAttributes

You can use the `GXGetInkAttributes` function to examine which attributes of an ink object are set.

```
gxInkAttribute GXGetInkAttributes(gxInk source);
```

`source` A reference to the ink to find the attributes of.

function result The ink attributes of the source ink object.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`ink_is_nil`

SEE ALSO

Ink attributes are described in the section “Ink Attributes” beginning on page 5-9.

To change the attributes of an ink object, use the `GXSetInkAttributes` function, described in the next section.

To examine the attributes of the ink object associated with a specified shape, use the `GXGetShapeInkAttributes` function, described on page 5-62.

GXSetInkAttributes

You can use the `GXSetInkAttributes` function to set or clear the attributes of an ink object.

```
void GXSetInkAttributes(gxInk target, gxInkAttribute attributes);
```

`target` A reference to the ink object to change the attributes of.

`attributes` The new ink attributes to be assigned.

DESCRIPTION

The `GXSetInkAttributes` function sets the attributes of the ink object referenced in the `target` parameter to those specified in the `attributes` parameter. If you pass `gxNoAttributes` for the `attributes` parameter, all attributes are cleared.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`

`ink_is_nil`

`parameter_out_of_range` (debugging version)

Notices (debugging version)

`attributes_already_set`

SEE ALSO

Ink attributes are described in the section “Ink Attributes” beginning on page 5-9.

To examine the attributes of an ink object, use the `GXGetInkAttributes` function, described in the previous section.

To change the attributes of the ink object associated with a specified shape, use the `GXSetShapeInkAttributes` function, described on page 5-63.

GXGetShapeInkAttributes

You can use the `GXGetShapeInkAttributes` function to examine the attributes of the ink associated with a shape.

```
gxInkAttribute GXGetShapeInkAttributes(gxShape source);
```

`source` A reference to the shape whose ink object you want the attributes of.

function result The attributes of the ink object associated with the source shape object.

ERRORS, WARNINGS, AND NOTICES**Errors**

out_of_memory
shape_is_nil

SEE ALSO

To set the attributes of the ink object associated with a shape, use the `GXSetShapeInkAttributes` function, described next.

To examine the attributes of an ink object itself, use the `GXGetInkAttributes` function, described on page 5-61.

Ink attributes are described in the section “Ink Attributes” beginning on page 5-9.

GXSetShapeInkAttributes

You can use the `GXSetShapeInkAttributes` function to set or clear attributes of the ink associated with a shape.

```
void GXSetShapeInkAttributes(gxShape target,
                             gxInkAttribute attributes);
```

`target` A reference to the shape whose ink object you want to change the attributes of.

`attributes` The new ink attributes to be assigned.

DESCRIPTION

The `GXSetShapeInkAttributes` function assigns the ink attributes specified by the `attributes` parameter to the ink object associated with the shape referenced in the `target` parameter. It is almost equivalent to the following call:

```
GXSetInkAttributes(GXGetShapeInk(target), attributes);
```

The only difference is that, if the source shape’s ink object is shared with other shapes, `GXSetShapeInkAttributes` creates a new copy of the ink object, attaches it to the source shape, and changes the attributes of the copy. That way, calling this function does not produce side effects on other shapes.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`shape_is_nil`
`parameter_out_of_range` (debugging version)

Notices (debugging version)

`attributes_already_set`

SEE ALSO

To examine the attributes of the ink object associated with a shape, use the `GXGetShapeInkAttributes` function, described in the previous section.

To set the attributes of an ink object itself, use the `GXSetInkAttributes` function, described on page 5-62.

Ink attributes are described in the section “Ink Attributes” beginning on page 5-9. The `GXSetInkAttributes` function is described on page 5-62.

The `GXGetShapeInk` function is described in the chapter “Shape Objects” in this book.

GXGetInkOwners

You can use the `GXGetInkOwners` function to determine the number of references to a particular ink object.

```
long GXGetInkOwners(gxInk source);
```

`source` A reference to the ink to find the owner count of.

function result The owner count of the source ink object.

DESCRIPTION

The `GXGetInkOwners` function returns as its function result the current number of references to the ink object.

ERRORS, WARNINGS, AND NOTICES**Errors**

`ink_is_nil`

SEE ALSO

Owner counts for ink objects are discussed in the section “Copying, Comparing, and Cloning Ink Objects” beginning on page 5-39, and in the section “Manipulating an Ink Object’s Owner Count” beginning on page 5-41.

GXGetInkTags

You can use the `GXGetInkTags` function to examine one or more of the tag objects associated with an ink object.

```
long GXGetInkTags(gxInk source, long tagType, long index,
                  long count, gxTag items[]);
```

<code>source</code>	A reference to the ink object whose tag list you want to examine.
<code>tagType</code>	The type of tag object to search for. A value of 0 indicates that you want to look for all tag types.
<code>index</code>	The (1-based) index of the first such tag reference to return.
<code>count</code>	The number of tag references to return.
<code>items</code>	An array to hold the returned tag references.

function result The number of tag references found that fit the criteria.

DESCRIPTION

The `GXGetInkTags` function searches the tag list of the `source` ink object for references to tag objects with the tag type specified by the `tagType` parameter. If you specify 0 for the `tagType` parameter, the `GXGetInkTags` function searches all tag types.

You can use the `index` and the `count` parameters to specify which tag references of the appropriate type the `GXGetInkTags` function should return. The `index` parameter indicates the first tag reference to return and the `count` parameter indicates how many tag references to return. The `index` parameter must be greater than 0. The `count` parameter must be greater than 0 or equal to the `gxSelectToEnd` constant (-1), which indicates that all tag references (starting with the tag reference indicated by the `index` parameter) should be returned.

The function result is the number of tag references found that fit the criteria. If you pass a value other than `nil` for the `items` parameter, the `GXGetInkTags` function returns in the `items` parameter the tag references that were found.

Typically, you call this function once with a `nil` value for the `items` parameter to determine the number of matching tag references. Then you allocate an appropriately sized array and call the function a second time to obtain the references themselves.

ERRORS, WARNINGS, AND NOTICES**Errors**

```

out_of_memory
ink_is_nil
index_is_less_than_one    (debugging version)
count_is_less_than_one    (debugging version)

```

Warnings

```

index_out_of_range
count_out_of_range

```

SEE ALSO

Tag objects are discussed in the chapter “Tag Objects” in this book.

To change the set of tag references associated with an ink, use the `GXSetInkTags` function, described in the next section.

GXSetInkTags

You can use the `GXSetInkTags` function to add, remove, or replace tag objects associated with an ink object.

```

void GXSetInkTags(gxInk target, long tagType, long index,
                 long oldCount, long newCount,
                 const gxTag items[]);

```

<code>target</code>	A reference to the ink object whose tag list you want to alter.
<code>tagType</code>	The type of tag objects to replace. A value of 0 indicates that you want to replace tags of all types.
<code>index</code>	The (1-based) index of the first tag reference (to a tag object of the appropriate type) to replace.
<code>oldCount</code>	The number of tag references to replace. A value of 0 specifies that you want to insert tag references before the tag reference indicated by the <code>index</code> parameter, rather than replace tag references. A value of -1 (the <code>gxSelectToEnd</code> constant) specifies that all tag references of the requested type, starting with the tag reference indicated by the <code>index</code> parameter, should be replaced.
<code>newCount</code>	The number of tag references to insert. A value of 0 specifies that there are no tag references to insert; the existing tag references that match the criteria you specify are removed from the <code>source</code> shape’s tag list and disposed of.
<code>items</code>	An array of tag references to insert in the tag list.

DESCRIPTION

The `GXSetInkTags` function allows you add tag references to an ink object's tag list, to remove tag references from the list, or to replace tag references in the list with new tag references. In any of these three cases, the `target` parameter specifies the ink object to be modified, the `newCount` parameter specifies the number of tag references to add, and the `items` parameter provides the new tag references.

- n To add tag references, set the `oldCount` parameter to 0. Use the `tagType` and the `index` parameters to specify where to add the new tag references. (For example, if you specify `nil` for the `tagType` parameter and 1 for the `index` parameter, this function inserts the new tag references before the current tag references. If you specify a value other than `nil` for the `tagType` parameter and a value of 2 for the `index` parameter, the function inserts the new tag references before the second tag reference with a tag type matching the `tagType` parameter.)
- n To remove tag references, set the `newCount` parameter to 0 and the `items` parameter to `nil`. You can use the `index` and the `oldCount` parameters to specify which tag references (of the specified type) should be removed. The `index` parameter indicates the first tag reference (of the specified type) to remove and the `oldCount` parameter indicates how many tag references (of the specified type) to remove.
- n To replace tag references, use the `tagType`, `index`, and `oldCount` parameters to indicate which tag references to replace, and use the `newCount` and `items` parameters to specify the new tag references to add. If `newCount` is greater than `oldCount`, the extra tag references are placed immediately adjacent to the last tag reference replaced.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>out_of_memory</code>	
<code>ink_is_nil</code>	
<code>tag_is_nil</code>	
<code>parameter_is_nil</code>	(debugging version)
<code>inconsistent_parameters</code>	(debugging version)
<code>parameter_out_of_range</code>	(debugging version)
<code>index_is_less_than_zero</code>	(debugging version)
<code>cannot_dispose_locked_tag</code>	(debugging version)

Warnings

`index_out_of_range`
`count_out_of_range`

Notices (debugging version)

`tag_already_set`

SEE ALSO

Tag objects are discussed in the chapter “Tag Objects” in this book.

To examine the set of tag references associated with an ink, use the `GXGetInkTags` function, described in the previous section.

Getting and Setting an Ink's Color

The functions described in this section allow you to get and set the color structure from a specified ink object, or from the ink object attached to a specified shape.

GXGetInkColor

You can use the `GXGetInkColor` function to examine the color of an ink object.

```
gxColor *GXGetInkColor(gxInk source, gxColor *data);
```

`source` A reference to the ink whose color you want.

`data` A pointer to a color structure. On return, the structure contains the color of the ink object.

function result The color of the source ink object.

DESCRIPTION

The `GXGetInkColor` function returns, as its function result and in the structure pointed to by the `data` parameter, the color of the ink referenced in the `source` parameter.

If the ink object reference or the pointer to the color structure is `nil`, an error is posted, and `nil` is returned as the function result.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
ink_is_nil
color_is_nil
```

SEE ALSO

Ink colors are introduced in the section “Color” beginning on page 5-7, and described fully in the chapter “Colors and Color-Related Objects” in this book.

To assign a color to an ink object, use the `GXSetInkColor` function, described next.

To examine the color of the ink object associated with a shape, use the `GXGetShapeColor` function, described on page 5-70.

GXSetInkColor

You can use the `GXSetInkColor` function to assign a color to an ink object.

```
void GXSetInkColor(gxInk target, const gxColor *data);
```

`target` A reference to the ink to assign the color to.

`data` A pointer to a color structure containing the color to assign to the ink.

DESCRIPTION

The `GXSetInkColor` function assigns the color pointed to by the `data` parameter to the ink object referenced in the `target` parameter. If the color references a color set or color profile object, QuickDraw GX increases the owner count of the referenced object.

SPECIAL CONSIDERATIONS

If the color space of the color pointed to by the `data` parameter is `gxNoSpace`, this function posts a `colorSpace_out_of_range` error.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`

`ink_is_nil`

`color_is_nil`

`colorSpace_out_of_range`

(debugging version)

`colorSet_access_restricted`

(debugging version)

`colorProfile_access_restricted`

(debugging version)

Notices (debugging version)

`color_already_set`

SEE ALSO

Ink colors are introduced in the section “Color” beginning on page 5-7, and described fully in the chapter “Colors and Color-Related Objects” in this book.

To examine the color of an ink object, use the `GXGetInkColor` function, described in the previous section.

To assign a color to the ink object associated with a shape, use the `GXSetShapeColor` function, described on page 5-71.

GXGetShapeColor

You can use the `GXGetShapeColor` function to examine the color of an ink object associated with a shape.

```
gxColor *GXGetShapeColor(gxShape source, gxColor *data);
```

`source` A reference to the shape whose ink you want the color of.

`data` A pointer to a color structure. On return, the structure contains the color of the ink object associated with the shape.

function result The color of the ink object associated with the source shape object.

DESCRIPTION

The `GXGetShapeColor` function returns, as its function result and in the structure pointed to by the `data` parameter, the color of the ink object associated with the shape object referenced in the `source` parameter.

This call is equivalent to

```
myColor = GXGetInkColor(GXGetShapeInk(myShape), myColor);
```

If the shape object reference or the pointer to the color structure is `nil`, an error is posted, and `nil` is returned as the function result.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
shape_is_nil
color_is_nil
```

SEE ALSO

To assign a color to the ink object associated with a shape, use the `GXSetShapeColor` function, described next.

To examine the color of an ink object directly, use the `GXGetInkColor` function, described on page 5-68.

Ink colors are introduced in the section “Color” beginning on page 5-7, and described fully in the chapter “Colors and Color-Related Objects” in this book.

The `GXGetShapeInk` function is described in the chapter “Shape Objects” in this book.

GXSetShapeColor

You can use the `GXSetShapeColor` function to assign a color to the ink object associated with a shape.

```
void GXSetShapeColor(gxShape target, const gxColor *data);
```

`target` A reference to the shape whose ink object you want to assign the color to.
`data` A pointer to a color structure containing the color to assign to the shape's ink object.

DESCRIPTION

The `GXSetShapeColor` function assigns the color pointed to by the `data` parameter to the ink object associated with the shape referenced in the `target` parameter.

Calling this function is almost equivalent to

```
GXSetInkColor(GXGetShapeInk(myShape), theColor);
```

except that, if the source shape's ink object is shared with other shapes, `GXSetShapeColor` creates a new copy of that ink object and attaches it to the source shape before changing its color. That way, calling this function does not produce any side effects on other shapes.

If the color pointed to by the `data` parameter references a color set or color profile object, QuickDraw GX increases the owner count of the referenced object.

SPECIAL CONSIDERATIONS

If you use this function to try to assign a color to a bitmap shape, the function posts an `illegal_type_for_shape` error. If the color space of the color pointed to by the `data` parameter is `gxNoSpace`, the function posts a `colorSpace_out_of_range` error.

ERRORS, WARNINGS, AND NOTICES

Errors

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>color_is_nil</code>	
<code>colorSpace_out_of_range</code>	(debugging version)
<code>colorSet_access_restricted</code>	(debugging version)
<code>colorProfile_access_restricted</code>	(debugging version)
<code>illegal_type_for_shape</code>	(debugging version)

Notices (debugging version)

`color_already_set`
`color_is_nil`

SEE ALSO

To examine the color of the ink object associated with a shape, use the `GXGetShapeColor` function, described in the previous section.

To assign a color to an ink object directly, use the `GXSetInkColor` function, described on page 5-69.

Ink colors are introduced in the section “Color” beginning on page 5-7, and described fully in the chapter “Colors and Color-Related Objects” in this book. Assigning colors to bitmaps is discussed in the bitmap shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*.

The `GXGetShapeInk` function is described in the chapter “Shape Objects” in this book.

Getting and Setting an Ink’s Transfer Mode

The functions described in this section allow you to get and set the transfer mode structure from a specified ink object, or from the ink object attached to a specified shape.

GXGetInkTransfer

You can use the `GXGetInkTransfer` function to examine the transfer mode of an ink object.

```
gxTransferMode *GXGetInkTransfer(gxInk source, gxTransferMode
*data);
```

`source` A reference to the ink whose transfer mode you want.

`data` A pointer to a transfer mode structure. On return, the structure contains the transfer mode of the ink object.

function result The transfer mode of the source ink object.

DESCRIPTION

The `GXGetInkTransfer` function returns, as its function result and in the structure pointed to by the `data` parameter, the transfer mode of the ink referenced in the `source` parameter.

If the ink object reference or the pointer to the transfer mode structure is `nil`, an error is posted, and `nil` is returned as the function result.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`ink_is_nil`
`transferMode_is_nil`

SEE ALSO

Transfer modes are described in the sections “About Transfer Modes” beginning on page 5-11, and “Working With Transfer Modes” beginning on page 5-44.

To assign a transfer mode to an ink object, use the `GXSetInkTransfer` function, described next.

To examine the transfer mode of the ink object associated with a shape, use the `GXGetShapeTransfer` function, described on page 5-74.

GXSetInkTransfer

You can use the `GXSetInkTransfer` function to assign a transfer mode to an ink object.

```
void GXSetInkTransfer(gxInk target, const gxTransferMode *data);
```

`target` A reference to the ink to assign the transfer mode to.

`data` A pointer to a transfer mode structure containing the transfer mode to assign to the ink.

DESCRIPTION

The `GXSetInkTransfer` function assigns the transfer mode pointed to by the `data` parameter to the ink object referenced in the `target` parameter.

SPECIAL CONSIDERATIONS

The color space of the transfer mode pointed to by the `data` parameter cannot be `gxNoSpace` or any of the packed color spaces (such as, for example, `gxRGB16Space`). If you specify `gxHighlightMode` in some but not all components of the transfer mode, this function posts an `inconsistent_parameters` error.

ERRORS, WARNINGS, AND NOTICES**Errors**

out_of_memory	
ink_is_nil	
transferMode_is_nil	
parameter_out_of_range	(debugging version)
inconsistent_parameters	(debugging version)
invalid_transferMode_colorSpace	(debugging version)
colorSpace_out_of_range	(debugging version)
colorSet_access_restricted	(debugging version)
colorProfile_access_restricted	(debugging version)

SEE ALSO

Transfer modes are described in the sections “About Transfer Modes” beginning on page 5-11, and “Working With Transfer Modes” beginning on page 5-44.

To examine the transfer mode of an ink object, use the `GXGetInkTransfer` function, described in the previous section.

To assign a transfer mode to the ink object associated with a shape, use the `GXSetShapeTransfer` function, described on page 5-75.

Color spaces are described in the chapter “Colors and Color-Related Objects” in this book.

GXGetShapeTransfer

You can use the `GXGetShapeTransfer` function to examine the transfer mode of the ink object associated with a shape.

```
gxTransferMode *GXGetShapeTransfer(gxShape source,
                                   gxTransferMode *data);
```

source A reference to the shape whose ink object you want the transfer mode of.

data A pointer to a transfer mode structure. On return, the structure contains the transfer mode of the shape’s ink object.

function result The transfer mode of the ink object associated with the source shape object.

DESCRIPTION

The `GXGetShapeTransfer` function returns, as its function result and in the structure pointed to by the `data` parameter, the transfer mode of the ink object associated with the shape referenced in the `source` parameter.

If the shape object reference or the pointer to the transfer mode structure is `nil`, an error is posted, and `nil` is returned as the function result.

This function is equivalent to

```
theMode = GXGetInkTransfer(GXGetShapeInk(myShape), theMode);
```

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`shape_is_nil`
`transferMode_is_nil`

SEE ALSO

Transfer modes are described in the sections “About Transfer Modes” beginning on page 5-11, and “Working With Transfer Modes” beginning on page 5-44.

To assign a transfer mode to the ink object associated with a shape, use the `GXSetShapeTransfer` function, described next.

To examine the transfer mode of an ink object directly, use the `GXGetInkTransfer` function, described on page 5-72.

The `GXGetShapeInk` function is described in the chapter “Shape Objects” in this book.

GXSetShapeTransfer

You can use the `GXSetShapeTransfer` function to assign a transfer mode to the ink object associated with a shape.

```
void GXSetShapeTransfer(gxShape target, const gxTransferMode
*data);
```

`target` A reference to the shape whose ink object you want to assign the transfer mode to.

`data` A pointer to a transfer mode structure containing the transfer mode to assign to the shape’s ink.

DESCRIPTION

The `GXSetShapeTransfer` function assigns the transfer mode pointed to by the `data` parameter to the ink object associated with the shape referenced in the `target` parameter.

Calling this function is almost equivalent to:

```
GXSetInkTransfer ( GXGetShapeInk ( myShape ) , theMode ) ;
```

except that, if the source shape's ink object is shared with other objects, `GXSetShapeTransfer` creates a new copy of that ink object and assigns it to the shape before changing its transfer mode. That way, calling this function does not produce any side effects on other shapes.

SPECIAL CONSIDERATIONS

The color space of the transfer mode pointed to by the `data` parameter cannot be `gxNoSpace` or any of the packed color spaces (such as, for example, `gxRGB16Space`). If you specify `gxHighlightMode` in some but not all components of the transfer mode, this function posts an `inconsistent_parameters` error.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>transferMode_is_nil</code>	
<code>parameter_out_of_range</code>	(debugging version)
<code>inconsistent_parameters</code>	(debugging version)
<code>invalid_transferMode_colorSpace</code>	(debugging version)
<code>colorSet_access_restricted</code>	(debugging version)
<code>colorSpace_out_of_range</code>	(debugging version)
<code>colorProfile_access_restricted</code>	(debugging version)

SEE ALSO

Transfer modes are described in the sections “About Transfer Modes” beginning on page 5-11, and “Working With Transfer Modes” beginning on page 5-44.

To examine the transfer mode of the ink object associated with a shape, use the `GXGetShapeTransfer` function, described in the previous section.

To assign a transfer mode to an ink object directly, use the `GXSetInkTransfer` function, described on page 5-73.

The `GXGetShapeInk` function is described in the chapter “Shape Objects” in this book.

Summary of Ink Objects

Constants and Data Types

The Ink Object

```
typedef struct gxPrivateInkRecord *gxInk;
```

Ink Attributes

```
enum gxInkAttributes{
    gxPortAlignDitherInk = 0x0001,
    gxForceDitherInk     = 0x0002,
    gxSuppressDitherInk  = 0x0004,
    gxSuppressHalftoneInk= 0x0008
};
```

```
typedef long gxInkAttribute;
```

The Color Structure

```
struct gxColor{
    gxColorSpace space;
    gxColorProfile profile;
    union {
        gxCMYKColor      cmyk;
        gxRGBColor       rgb;
        gxRGBAColor      rgba;
        gxHSVColor       hsv;
        gxHLSColor       hls;
        gxCIEColor       cie;
        gxYIQColor       yiq;
        gxColorValue     gray;
        gxGrayAColor     graya;
        unsigned short   pixel16;
        unsigned long    pixel32;
        gxIndexedColor   indexed;
        gxColorValue     component[4];
    } element;
};
```

```
typedef unsigned char gxComponentMode;
```

The Transfer Mode Structure

```

struct gxTransferMode{
    gxColorSpace          space;
    gxColorSet            set;
    gxColorProfile        profile;
    Fixed                 sourceMatrix[5][4];
    Fixed                 deviceMatrix[5][4];
    Fixed                 resultMatrix[5][4];
    gxTransferFlag        flags;
    struct gxTransferComponent component[4];
};

```

Transfer Mode Flags

```

enum gxTransferFlags{
    gxRejectSourceTransfer = 0x0001, /* At least one source component
                                       must be out of range */
    gxRejectDeviceTransfer = 0x0002, /* At least one device component
                                       must be out of range */
    gxSingleComponentTransfer= 0x0004 /* duplicate gxTransferComponent[0]
                                       for all components in transfer */
};

typedef long gxTransferFlag;

```

The Transfer Component Structure

```

struct gxTransferComponent{
    gxComponentMode    mode;
    gxComponentFlag    flags;
    gxColorValue       sourceMinimum;
    gxColorValue       sourceMaximum;
    gxColorValue       deviceMinimum;
    gxColorValue       deviceMaximum;
    gxColorValue       clampMinimum;
    gxColorValue       clampMaximum;
    gxColorValue       operand;
};

```


Component Modes

```
enum gxComponentModes{
    gxNoMode = 0,
    gxCopyMode,
    gxAddMode,
    gxBlendMode,
    gxMigrateMode,
    gxMinimumMode,
    gxMaximumMode,
    gxHighlightMode,
    gxAndMode,
    gxOrMode,
    gxXorMode,
    gxRampAndMode,
    gxRampOrMode,
    gxRampXorMode,
    gxOverMode,
    gxAtopMode,
    gxExcludeMode,
    gxFadeMode
};
```

Transfer Component Flags

```
enum gxComponentFlags{
    gxOverResultComponent= 0x01, /* AND the result component with
                                   0xFFFF before clamping */
    gxReverseComponent    = 0x02 /* Reverse source and device components
                                   before applying transfer mode */
};

typedef unsigned char gxComponentFlag;
```

Functions

Creating and Manipulating Ink Objects

```
gxInk GXNewInk                (void);
void GXDisposeInk             (gxInk target);
gxInk GXCopyToInk             (gxInk target, gxInk source);
boolean GXEqualInk             (gxInk one, gxInk two);
gxInk GXCloneInk              (gxInk source);
```

Manipulating Ink Object Properties

```

void GXResetInk          (gxInk target);
gxInkAttribute GXGetInkAttributes
                        (gxInk source);
void GXSetInkAttributes  (gxInk target, gxInkAttribute attributes);
gxInkAttribute GXGetShapeInkAttributes
                        (gxShape source);
void GXSetShapeInkAttributes(gxShape target, gxInkAttribute attributes);
long GXGetInkOwners      (gxInk source);
long GXGetInkTags        (gxInk source, long tagType, long index,
                        long count, gxTag items[]);
void GXSetInkTags        (gxInk target, long tagType, long index,
                        long oldCount, long newCount,
                        const gxTag items[]);

```

Getting and Setting an Ink's Color

```

gxColor *GXGetInkColor   (gxInk source, gxColor *data);
void GXSetInkColor       (gxInk target, const gxColor *data);
gxColor *GXGetShapeColor (gxShape source, gxColor *data);
void GXSetShapeColor     (gxShape target, const gxColor *data);

```

Getting and Setting an Ink's Transfer Mode

```

gxTransferMode *GXGetInkTransfer
                (gxInk source, gxTransferMode *data);
void GXSetInkTransfer      (gxInk target, const gxTransferMode *data);
gxTransferMode *GXGetShapeTransfer
                (gxShape source, gxTransferMode *data);
void GXSetShapeTransfer    (gxShape target, const gxTransferMode *data);

```

Transform Objects

Contents

About Transform Objects	6-5
Transform Object Properties	6-6
Clip	6-7
Mapping	6-10
View Port List	6-11
Hit-Test Parameters	6-11
Default Transform Objects	6-14
Using Transform Objects	6-15
Creating and Manipulating Transform Objects	6-15
Creating and Disposing of Transform Objects	6-15
Copying, Comparing, and Cloning Transform Objects	6-16
Implicit Creation of Transform Objects	6-18
Loading and Unloading Transform Objects	6-18
Manipulating Transform Object Properties	6-19
Manipulating a Transform Object's Owner Count	6-19
Getting and Setting a Transform Object's Tag References	6-20
Resetting Default Transform Properties	6-20
Getting, Setting, and Modifying the Transform Clip	6-20
Moving, Scaling, Rotating, and Skewing Shapes	6-23
Modifying the Transform Mapping	6-24
Modifying Shape Geometry	6-26
Manipulating the View Port List	6-28
Setting Up Hit-Test Parameters	6-30
Transform Objects Reference	6-31
Constants and Data Types	6-31
The Transform Object	6-31
Shape Parts for Hit-Testing	6-32

Functions	6-32	
Creating and Manipulating Transform Objects		6-33
GXNewTransform	6-33	
GXDisposeTransform	6-34	
GXCopyToTransform	6-35	
GXEqualTransform	6-36	
GXCloneTransform	6-37	
Manipulating Transform Object Properties		6-38
GXResetTransform	6-38	
GXGetTransformOwners	6-39	
GXGetTransformTags	6-40	
GXSetTransformTags	6-41	
Getting and Setting the Clip		6-43
GXGetTransformClip	6-43	
GXSetTransformClip	6-44	
GXGetShapeClip	6-45	
GXSetShapeClip	6-46	
Performing Geometric Operations on Transform Clips		6-48
GXUnionTransform	6-49	
GXIntersectTransform	6-50	
GXDifferenceTransform	6-51	
GXReverseDifferenceTransform	6-52	
GXExcludeTransform	6-53	
Getting and Setting the Mapping		6-53
GXGetTransformMapping	6-54	
GXSetTransformMapping	6-55	
GXGetShapeMapping	6-56	
GXSetShapeMapping	6-57	
Transforming Shapes by Modifying Transform Mappings		6-58
GXMoveTransform	6-58	
GXMoveTransformTo	6-59	
GXScaleTransform	6-60	
GXRotateTransform	6-62	
GXSkewTransform	6-63	
GXMapTransform	6-64	
Transforming Shapes by Modifying Shape Geometries		6-65
GXMoveShape	6-66	
GXMoveShapeTo	6-67	
GXScaleShape	6-68	
GXRotateShape	6-70	
GXSkewShape	6-71	
GXMapShape	6-72	
Getting and Setting the View Port List		6-73
GXGetTransformViewPorts	6-73	
GXSetTransformViewPorts	6-74	
GXGetShapeViewPorts	6-75	
GXSetShapeViewPorts	6-76	

CHAPTER 6

Getting and Setting the Hit-Test Parameters	6-77
GXGetTransformHitTest	6-78
GXSetTransformHitTest	6-79
GXGetShapeHitTest	6-80
GXSetShapeHitTest	6-81
Summary of Transform Objects	6-82
Constants and Data Types	6-82
Functions	6-83

This chapter describes transform objects and the functions you can use to manipulate them. Read this chapter if you need to clip parts of a shape for drawing, modify the position or dimensions of a shape, modify the view ports to which a shape is drawn, or hit-test a shape.

Before reading this chapter, you should be familiar with the information in the chapter “Introduction to QuickDraw GX” in this book. You should also be familiar with shape objects, as discussed in the chapter “Shape Objects” in this book.

For specific information about how transform objects affect bitmap and picture shapes, see *Inside Macintosh: QuickDraw GX Graphics*. For specific information about how transform objects affect typographic shapes, see *Inside Macintosh: QuickDraw GX Typography*. For information about the mathematical foundation of transform mappings, see the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*. For information about view ports, see the chapter “View-Related Objects” in this book.

This chapter introduces QuickDraw GX transform objects and describes their properties. It then shows how to use QuickDraw GX functions to

- n create and manipulate transform objects
- n manipulate transform object properties, including the clip, view port list, and hit-test parameters
- n perform mapping operations to change the translation, rotation, scale, skew, or perspective of transform objects and shape objects

About Transform Objects

A transform object exists to modify, or *transform*, the appearance or behavior of a shape. Each QuickDraw GX shape consists of a shape object, a style object, an ink object, and a transform object; the transform object specifies where the shape is drawn, how its appearance is transformed when drawn, and how the user can interact with the drawn shape. You can think of a transform object as a filter between the shape object and its drawing destination of one or more view ports.

QuickDraw GX identifies an individual transform object through a transform **reference**. To obtain information about a transform object, you must send its reference as a parameter to a QuickDraw GX function (except that you can determine if two references identify the same transform object simply by comparing them for equality, and you can examine a reference to see if it is `nil`).

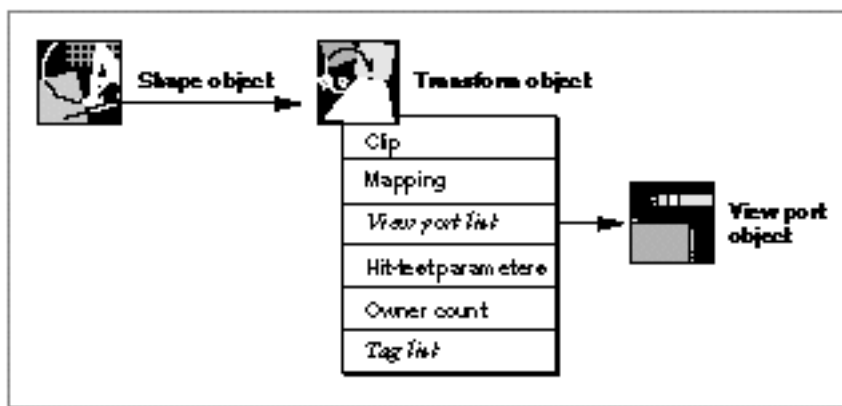
A shape (other than a picture shape) always refers to a single transform object. Several shapes, however, can refer to the same transform object. When they do, the transform object is shared and its transformations apply to all of those shapes. If you use a function that directly manipulates the transform object, the behavior of all shape objects associated with it changes.

Transform Object Properties

The interface to transform objects is entirely procedural. You manipulate the information in a transform object by modifying its properties using QuickDraw GX functions.

Transform objects have six accessible properties, as shown in Figure 6-1. Note that, because a transform is an object and not a data structure, the order of the properties shown in Figure 6-1 is completely arbitrary. Properties in italics are references to other objects.

Figure 6-1 The transform object and its properties



These are the six accessible properties in a transform object:

- n **Clip.** A reference to a specialized shape geometry that defines the visible area of the shape associated with this transform object. Only the parts of the shape that overlap with the clip remain visible when the shape is drawn. The transform clip is further described in the next section, “Clip.”
- n **Mapping.** A mathematical matrix that specifies the translation, scaling, rotation, skewing, and perspective of the shape associated with this transform object. The transform mapping is further described in the section “Mapping” on page 6-10.
- n **View port list.** An array of references to the view ports that the shape associated with this transform object can be drawn to. The view port list is further described in the section “View Port List” on page 6-11.
- n **Hit-test parameters.** Two values that provide criteria for hit-testing the shape associated with this transform object. The hit-test parameters specify what parts of the shape are to be tested for, and how close to a part a hit point must be to be considered successful. The hit-test parameters are further described in the section “Hit-Test Parameters” on page 6-11.

Transform Objects

- n **Owner count.** The number of existing references to this transform object. General information about owner counts is in the chapter “Introduction to QuickDraw GX” in this book; the section “Copying, Comparing, and Cloning Transform Objects” beginning on page 6-16 describes when and how to examine and alter a transform object’s owner count.
- n **Tag list.** A list of references to custom information about this transform object, stored in private data structures called tag objects. The chapter “Tag Objects” in this book describes tag objects in general and how you can use them to add custom information to objects.

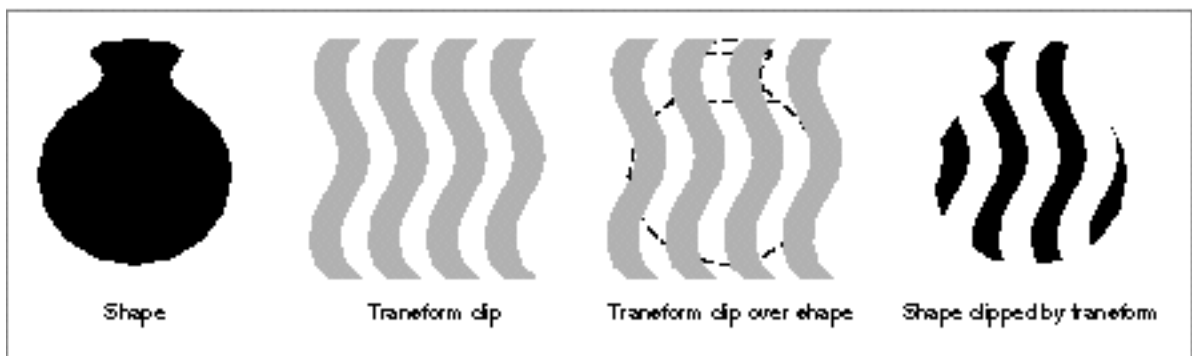
QuickDraw GX provides functions to manipulate each of these transform object properties.

Clip

The **clip** property of a transform object is a specialized shape geometry that functions as a mask to restrict the visibility of a shape when it is displayed or printed. The clip shape is equivalent to a **primitive shape**, a shape (of any type) whose geometry and fill properties by themselves define the shape. In other words, a primitive shape does not use any information from a style object or transform object to determine its location, dimension, or even pen thickness; all dimensional information about a primitive shape is in the shape object itself.

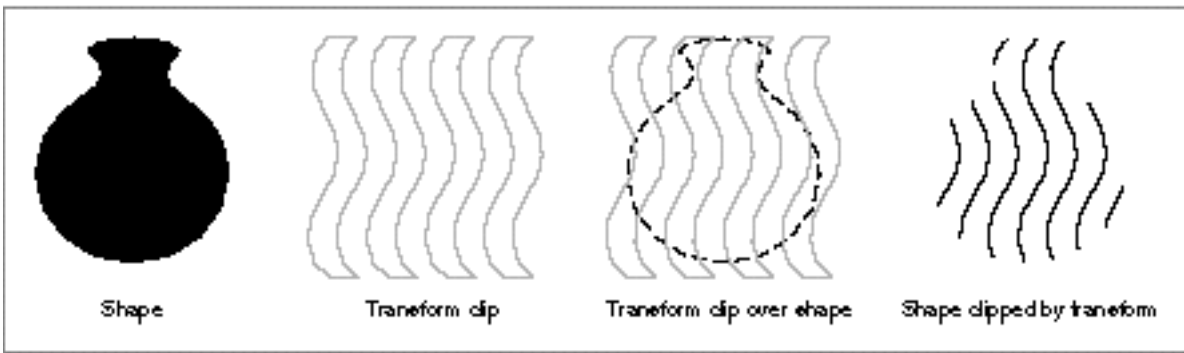
The filled or framed parts of a transform’s clip define the areas in which the shape attached to that transform show through. Figure 6-2 shows the effect of using a transform object to clip a shape representing a vase. The vase shape references a transform object whose clip property defines a clip shape consisting of four filled paths. Only the parts of the vase that intersect the filled paths are allowed to show through.

Figure 6-2 A transform clip



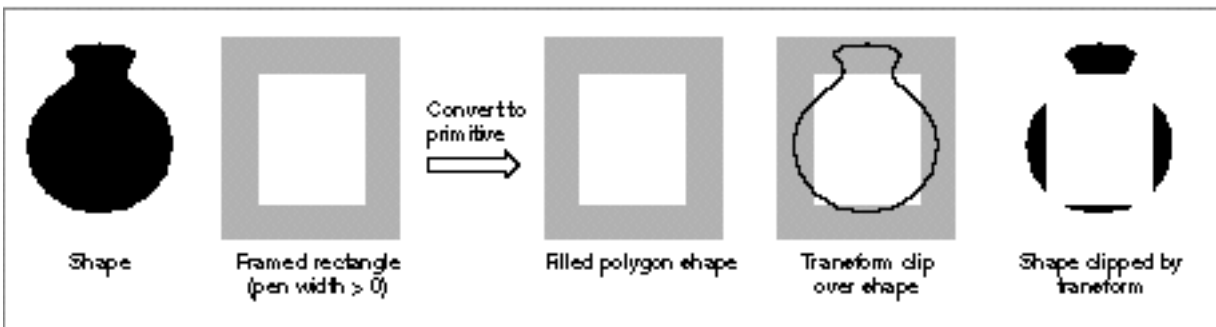
If the clip were a framed path shape instead of the filled path shape shown in Figure 6-2, only the parts of a shape that intersect the frame itself would be visible. And because the pen width is 0 for a primitive shape, the frame would be of hairline width only; the parts of the shape both outside and inside the hairline frame would be clipped out, as shown in Figure 6-3.

Figure 6-3 A framed transform clip

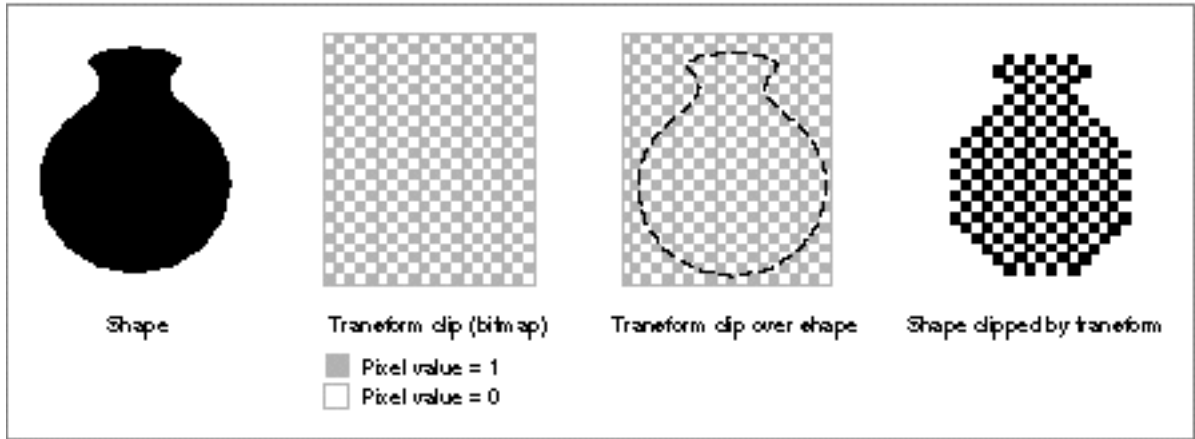


To use a framed shape with nonzero pen width as a clip shape, you first convert it to a primitive shape, at which point it becomes a filled shape in which the filled areas correspond to the pen width in the original framed shape. Figure 6-4 shows an example of converting a framed shape with a nonzero pen width into a transform clip shape.

Figure 6-4 Converting a framed shape with a nonzero pen width into a transform clip

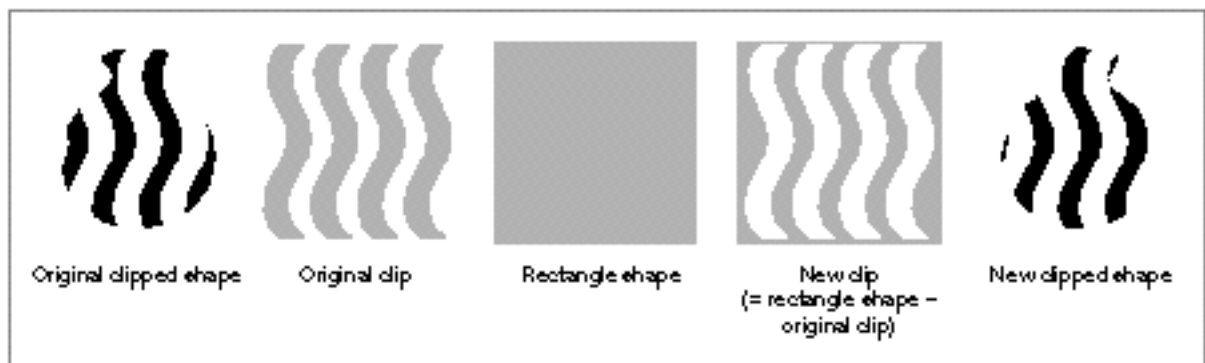


You can use a bitmap shape as a clip, but only if its pixel depth is 1—meaning that each pixel has a value of 0 or 1—and if its color profile is nil. When a transform clip is a bitmap, its individual pixels mask the shape that is drawn. A pixel with a value of 1 allows the shape geometry to show through the area covered by that pixel, and a pixel with a value of 0 clips out the part of the shape covered by that pixel. Figure 6-4 shows an example.

Figure 6-5 Using a bitmap as a transform clip

For more information on bitmap shapes, see the bitmap shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*. For more information on primitive shapes, see the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*.

QuickDraw GX provides functions that allow you to modify a transform clip by performing constructive geometry operations—such as union and intersection—between it and another shape. With these operations you can build a clip from one or more shapes. For example, you can start with the default clip shape, `gxFullShape`, which allows everything to show through, and use constructive geometry operations to restrict the visibility. Or you can start with an empty shape, `gxEmptyShape`, and use constructive geometry operations to increase visibility. Figure 6-6 shows one example (using reverse difference); the section “Getting, Setting, and Modifying the Transform Clip” beginning on page 6-20 describes the operations you can perform and the QuickDraw GX functions you use to modify a transform clip.

Figure 6-6 Modifying a transform clip by subtracting it from another shape

Mapping

The **mapping** property is a 3×3 matrix that specifies one or more transformations that a transform object performs on its associated shape. It is the transforming aspect of the mapping property that gives the transform object its name. You can use the transform mapping to perform the following operations on a shape:

- n translation, which changes the position of the shape
- n scaling, which shrinks or enlarges the shape horizontally or vertically or both
- n rotation, which turns the shape about a fixed point
- n skewing, which distorts the shape progressively along a single axis
- n perspective, which distorts the shape to provide a three-dimensional appearance

Figure 6-7 shows examples of some of these transformations.

Figure 6-7 Effects of the transform mapping



You can combine one or more of the possible transformations in a single mapping matrix. For example, you can specify 200 percent scaling and 30-degree rotation in the same mapping. The **identity mapping**, which is a matrix whose elements have the value 1.0 along the diagonal and 0.0 elsewhere, specifies no transformation. An identity mapping applied to a shape leaves it unchanged. The identity mapping is the default mapping for a transform.

One important advantage of having a mapping property separate from a shape's geometry is that you can change the visual appearance of a shape in many different ways and at many different times without ever changing the geometry of the shape itself. This minimizes the accumulation of errors, and also allows a set of identical shape geometries to result in many different appearances. QuickDraw GX provides functions with which you can easily modify the mapping of a transform object to perform translation, scaling, rotation, and skewing. See the section "Modifying the Transform Mapping" beginning on page 6-24 for more information and examples.

If you want to, however, you can also transform a shape by changing its geometry directly. Direct manipulation of shape geometry can be faster than modifying the transform object, and may be more appropriate when you want to change the fundamental nature of a shape. QuickDraw GX provides functions with which you

Transform Objects

can directly modify the geometry of a shape object to perform translation, scaling, rotation, and skewing. See the section “Modifying Shape Geometry” beginning on page 6-26 for more information and examples.

For general information about the characteristics and capabilities of mappings, see the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

View Port List

The **view port list** property is an array of references to view port objects. The list specifies all of the view ports through which the shape associated with this transform object can be drawn. A transform object must have at least one view port reference in its view port list if any drawing is to occur. If it has more than one view port in the list, drawing occurs to all of its view ports simultaneously.

You may want to have several view ports in a list if you want the shapes associated with a transform object to display in several windows, or in different places in the same window. If you implement offscreen drawing, your transform object can reference both onscreen and offscreen view ports.

The same view port can be in the view port list more than once. For example, you may wish to draw the same shape several times, using a transfer mode that accounts for the color already on the view device. The view port list controls the order in which a shape is drawn: the shape is drawn to the first view port in the list before it is drawn to the second one in the list, and so on. You can use that information to control the order in which colors are manipulated by the transfer mode you have chosen. For information about transfer modes, see the chapter “Ink Objects” in this book.

Like transform objects, view port objects have their own clip and mapping properties that affect how a shape appears when drawn. So also do view device objects. Note that view ports do not correspond to view devices; for example, you don’t need to have multiple view ports in the view port list property just because the computer has several screens. View ports and view devices are described in the chapter “View-Related Objects” in this book.

For further information and examples of manipulating the view port list of a transform object, see “Manipulating the View Port List” beginning on page 6-28.

Hit-Test Parameters

The **hit-test parameters** property of a transform object consists of two values that specify the criteria to be used for hit-testing shapes associated with this transform object. Hit-testing itself is introduced in the chapter “Introduction to QuickDraw GX” and described in more detail in the chapter “Shape Objects,” in this book.

The hit-testing functions `GXHitTestShape` and `GXHitTestPicture` use the hit-test parameters in the transform object. The hit-testing parameters consist of

- n a mask that specifies the parts of a shape that are to be tested for a hit
- n a value that specifies the tolerance, or distance from any of the parts, that a hit point can be and still be considered to represent a successful hit

Shape-Parts Mask

The parts of a shape object that you can consider in hit-testing are shown in Table 6-1. They are defined in the `gxShapeParts` enumeration.

Table 6-1 Shape parts for hit-testing, from the `gxShapeParts` enumeration

Constant	Value	Explanation
<code>gxNoPart</code>	0	Not in any part of the shape. This value is returned by a hit-testing function if no shape part was successfully hit.
<code>gxBoundsPart</code>	0x0001	Anywhere within the bounding rectangle of the shape.
<code>gxGeometryPart</code>	0x0002	Anywhere within the geometry of the shape. If the shape is framed, this includes just the curves or lines that make up the contours; if the shape is filled, this includes all filled areas.
<code>gxPenPart</code>	0x0004	Anywhere in the pen swath. For example, if this shape's style object has its <code>gxCenterFrameStyle</code> attribute set, this includes anywhere within half the pen width on either side of all curves or lines that make up the contours.
<code>gxCornerPointPart</code>	0x0008	On any geometric (on-curve) point in the shape geometry
<code>gxControlPointPart</code>	0x0010	On any (off-curve) control point in the shape geometry.
<code>gxEdgePart</code>	0x0020	On the edge of the geometry; along any of the curves or lines that make up the contours.
<code>gxJoinPart</code>	0x0040	Within the geometry of a join that is part of the shape.
<code>gxStartCapPart</code>	0x0080	Within the geometry of a start cap that is part of the shape.
<code>gxEndCapPart</code>	0x0100	Within the geometry of an end cap that is part of the shape.
<code>gxDashPart</code>	0x0200	Within the geometry of a dash element that is part of the shape.
<code>gxPatternPart</code>	0x0400	Within the geometry of a pattern element that is part of the shape.
<code>gxGlyphBoundsPart</code>	0x0040	(Same value as <code>gxJoinPart</code> .) For a typographic shape, anywhere within the bounding rectangle of an individual glyph.

Table 6-1 Shape parts for hit-testing, from the `gxShapeParts` enumeration (continued)

Constant	Value	Explanation
<code>gxGlyphFirstPart</code>	<code>0x0080</code>	(Same value as <code>gxStartCapPart</code> .) For a typographic shape with horizontal text, anywhere within the left half of a glyph, including its left side bearing; for vertical text, anywhere in the top half of the glyph. “Left half” or “top half” means half the advance width: half of the distance from the left margin of the left side bearing to the right margin of the right side bearing.
<code>gxGlyphLastPart</code>	<code>0x0100</code>	(Same value as <code>gxEndCapPart</code> .) For a typographic shape with horizontal text, anywhere within the right half of a glyph, including its right side bearing; for vertical text, anywhere in the bottom half of the glyph. “Right half” or “bottom half” means half the advance width: half of the distance from the right margin of the right side bearing to the left margin of the left side bearing.
<code>gxSideBearingPart</code>	<code>0x0200</code>	(Same value as <code>gxDashPart</code> .) For a typographic shape, within a glyph side bearing.
<code>gxAnyPart</code>	<code>0x07FF</code>	Any of the above parts. You can pass this value to a hit-testing function if you want it to test for all possible shape parts.

NOTE Points, control points, contours, joins, caps, dashes, patterns, and other components of geometric shape geometries are described in the geometric shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*. The `gxCenterFrameStyle` attribute and other style attributes that apply to geometric shapes are described in the geometric styles chapter of *Inside Macintosh: QuickDraw GX Graphics*.

Glyph bounding rectangles, side bearings, and advance widths are described in the introductory chapter of *Inside Macintosh: QuickDraw GX Typography*.

Tolerance

The tolerance is a value that describes how close the tested point must be to a shape part before that part is considered to have been hit. Tolerance is specified in the shape object’s geometry space (except that style-object information is included when testing for pen, joins, caps, dashes, patterns, and typographic shapes). The tolerance is dimensionless (it has no metric, such as inches), and can have any fixed-point value in the range of $-32,767.0$ to approximately $32,768.0$. A tolerance of 0 means that the hit point must exactly coincide with a shape part for a successful hit.

For more information about coordinate spaces, see the chapter “View-Related Objects” in this book.

Setting Up the Parameters

Before calling `GXHitTestShape` or `GXHitTestPicture`, you set up the shape-parts mask in the transform object to specify the shape parts you are interested in testing for. Note that values specifying join, cap, and dash parts in geometric shapes are used in typographic shapes to specify various glyph parts instead. Note also that you can specify no parts or all parts in the mask. You also specify a tolerance, which should be 0 if the hit point must exactly coincide with a shape part for a successful hit. See “Setting Up Hit-Test Parameters” beginning on page 6-30 for more information and examples.

The `GXHitTestShape` function is described in the chapter “Shape Objects” in this book, with additional information for typographic shapes and typographic shape parts in the typographic shapes chapter of *Inside Macintosh: QuickDraw GX Typography*. The `GXHitTestPicture` function is described in the picture shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*.

Default Transform Objects

QuickDraw GX provides a default transform object for you. When you first create a shape, that shape’s transform reference is `nil`, which means that QuickDraw GX uses the default transform object as that shape’s transform. (This assumes that you have not modified the default shape object so that it references a specific transform; see the discussion of default shapes in the chapter “Shape Objects” in this book.)

Also, when you explicitly create a transform object, it is initially a copy of the default transform. These are the properties of the default transform object:

- A clip shape that is a full shape. No clipping occurs when QuickDraw GX draws the shape associated with this transform; the clip has no effect.
- A mapping that is the identity mapping. When drawing, QuickDraw GX does not change the position, scaling, skewing, rotation, or perspective of the shape associated with this transform; the mapping has no effect.
- A view port list that contains a single default view port that covers all screen view devices. See the chapter “View-Related Objects” in this book for more information on the default view port.
- Hit-test parameters that are
 - the default shape-parts mask, in which only the `gxBoundsPart` shape part is specified. QuickDraw GX considers the hit successful if the test point falls within the bounding rectangle of the shape associated with this transform.
 - the default hit-test tolerance, which is 0. The test point cannot be any distance outside of the bounding rectangle if the hit is to be considered successful.
- An owner count of 1.
- An empty tag list.

QuickDraw GX provides a function that allows you to reset a transform object to these default values at any time. See the section “Manipulating a Transform Object’s Owner Count” on page 6-19 for more information.

Using Transform Objects

This section describes the basic transform-creation and transform-manipulation capabilities that QuickDraw GX provides. It describes how you can

- n create and manipulate transform objects
- n manipulate the common transform object properties
- n use geometric operations to modify a transform's clip property
- n transform a shape by modifying its transform object's mapping
- n transform a shape by modifying its geometry
- n manipulate a transform's view port list
- n set up hit-testing parameters

Creating and Manipulating Transform Objects

This section describes how you can create and interact with transform objects as whole entities—to create, dispose of, copy, compare, and clone them. Manipulating the individual properties of transform objects is described under “Copying, Comparing, and Cloning Transform Objects” beginning on page 6-16 and in subsequent sections.

Creating and Disposing of Transform Objects

QuickDraw GX provides the `GXNewTransform` function to allow you to create a new transform object. You can also create a new transform object by copying an existing one with the `GXCopyToTransform` function. Once you have created a transform object, you can modify its properties using functions such as those described in the section “Copying, Comparing, and Cloning Transform Objects” beginning on page 6-16.

You need to explicitly create a transform object if you want a single nondefault transform to apply to several shapes. You can also indirectly cause the creation of a new transform object by modifying a transform property under certain conditions; see “Implicit Creation of Transform Objects” on page 6-18 for more information.

To delete your application's reference to a transform object, call the `GXDisposeTransform` function, which may or may not actually release the memory allocated for that transform object, depending on the transform's owner count. The function decreases the transform object's owner count by 1; if that brings the owner count to zero, the transform is completely deleted and its memory released. For more discussion of transform-object owner counts, see “Manipulating a Transform Object's Owner Count” on page 6-19.

Listing 6-1 is a code fragment that creates a transform object (`myTransform`) and assigns it to a shape object (`myRectangle`).

Listing 6-1 Creating and disposing of a transform object

```

gxTransform    myTransform;
gxShape        myRectangle;
myTransform =  GXNewTransform();
.
.  /* set the transform object's properties (not shown) */
.
myRectangle =  GXNewRectangle(gxRectangleType);
GXSetShapeTransform(myRectangle, myTransform);
GXDisposeTransform(myTransform);

```

Notice that the code disposes of the `myTransform` reference to the transform object immediately after it is assigned to the shape. The code no longer needs the reference, and this decreases the transform's owner count, allowing it to be deleted as soon as the shape no longer needs it. The proper place to call `GXDisposeTransform` is immediately after you have finished using a transform in your code, even if you know that another object's use prevents the transform from being deleted at that time.

The `GXNewTransform` function is described on page 6-33. The `GXDisposeTransform` function is described on page 6-34.

Copying, Comparing, and Cloning Transform Objects

You can use the `GXCopyToTransform` function to copy information from one transform object to another or to create a new copy of a transform object.

You can test if two transform-object references refer to the same transform object by simply testing the references for equality. You can also compare two different transform objects for equality with the `GXEqualTransform` function. For two transform objects to be equal, their clips, mappings, view port lists, and hit-test parameters must have identical values, although their owner count and tag list do not need to be identical. Transform object copies created with the `GXCopyToTransform` function are always equal, by these criteria, to the transform from which they were copied.

The following code fragment creates a copy (`lineTransform`) of the transform object associated with the default line shape. It then scales the line shape by changing its transform mapping. Finally, it recopies the original transform back into `lineTransform` to restore the unscaled values.

Transform Objects

```

gxTransform    lineTransform, savedTransform;
gxShape        defaultLine;
defaultLine = GXGetDefaultShape(gxLineType);
lineTransform = GXGetShapeTransform(defaultLine);
savedTransform = GXCopyToTransform(nil, lineTransform);
GXScaleTransform(lineTransform, ff(2), ff(2), 0, 0);
.
. /* use the scaled transform (not shown) */
.
GXCopyToTransform(lineTransform, savedTransform);
GXDisposeTransform(savedTransform);

```

Note that the first call to `GXCopyToTransform` in the above code creates a new transform object, whereas the second call causes the contents of one transform to be copied into another.

In certain circumstances, you may want to copy a reference to a transform object without actually copying the object itself. For example, you may want two variables to refer to the same transform object, so that editing one of them affects both. Or, you may want to preserve a reference to a transform so that it cannot be inadvertently deleted. This is called cloning a transform; you can use the `GXCloneTransform` function to clone a transform object.

Functionally, `GXCloneTransform` does nothing more than increase the owner count of a transform. The code in Listing 6-2 clones a shape's original transform object to preserve it from being deleted, changes the shape's transform object temporarily to perform several operations (not shown in the example), and then restores the original transform. In this example, the original transform object is called `saved` and the one that is used for the operations is called `newXform`.

Listing 6-2 Cloning a transform to prevent it from being deleted

```

gxTransform saved = GXGetShapeTransform(aShape);
GXCloneTransform(saved);
GXSetShapeTransform(aShape, newXform);
.
. /* use the new transform (not shown) */
.
GXSetShapeTransform(aShape, saved);

```

The `saved` transform object must be cloned because, in the process of associating a new transform object with a shape, the `GXSetShapeTransform` function decrements the owner count of the previously associated transform object. Cloning prevents the `saved` object from being deleted because cloning increments the owner count, which prevents the owner count of the `saved` transform object from going down to 0.

For more information about cloning objects, see the chapter “Introduction to Objects” in this book. For more information on manipulating transform owner counts, see the section “Manipulating a Transform Object’s Owner Count” beginning on page 6-19 of this chapter.

The `GXCopyToTransform` function is described on page 6-35. The `GXCloneTransform` function is described on page 6-37. The `GXEqualTransform` function is described on page 6-36.

The `GXScaleTransform` function is described on page 6-60. The `GXSetShapeTransform` function is described in the chapter “Shape Objects” in this book.

Implicit Creation of Transform Objects

QuickDraw GX provides two kinds of functions that modify transform properties:

- n The first kind, such as `GXSetTransformClip` and `GXSetTransformMapping`, takes a transform reference as a parameter; it directly alters a property of the transform and thus affects all shapes that use that transform. No new transform object is created when you call this kind of function.
- n The second kind, such as `GXSetShapeClip` and `GXSetShapeMapping`, takes a shape reference as a parameter; it alters a property of whatever transform object is used by that shape. To keep from inadvertently affecting other shapes that use the same transform, QuickDraw GX creates a copy of the transform object and modifies the copy if more than one shape object shares the transform.

In addition, if you call a function that normally affects only a shape’s geometry, such as `GXRotateShape`, and the shape’s `gxMapTransformShape` attribute is also set, the shape’s transform mapping is changed instead; in that case, if the transform is shared by more than one object, QuickDraw GX creates a copy and modifies the copy.

The `gxMapTransformShape` attribute is described in the chapter “Shape Objects” in this book. How it affects the functions described in this chapter is discussed in the section “Moving, Scaling, Rotating, and Skewing Shapes” beginning on page 6-23.

Loading and Unloading Transform Objects

Although you rarely need to, you can influence memory-allocation decisions involving objects that you have created. If your application needs to have a transform object in memory, you can force QuickDraw GX to load it into memory. When your application no longer needs the transform object in a loaded state, you can instruct QuickDraw GX to unload it.

You call the `GXLoadTransform` function to make sure that a transform object is in memory; if necessary, QuickDraw GX brings the object into memory from an unloaded state. You can call the `GXUnloadTransform` function to instruct QuickDraw GX that it is free to unload the transform object at any time. These functions are described in the memory management chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

Manipulating Transform Object Properties

This section describes how to manipulate the common object properties of transform objects: owner count and tag list. It also describes how to restore a transform object's properties to their default values.

To manipulate the clip of a transform, see the section “Getting, Setting, and Modifying the Transform Clip” beginning on page 6-20. To manipulate the mapping of a transform, see the section “Moving, Scaling, Rotating, and Skewing Shapes” beginning on page 6-23. To manipulate the view port list of a transform, see the section “Manipulating the View Port List” beginning on page 6-28. To manipulate the hit-test parameters of a transform, see the section “Setting Up Hit-Test Parameters” beginning on page 6-30.

For manipulating a transform object as a whole, see “Creating and Manipulating Transform Objects” beginning on page 6-15.

Manipulating a Transform Object's Owner Count

The owner count of an object indicates the number of current references to that object. In general, QuickDraw GX manages owner counts for you. For example, when you create a new transform object, QuickDraw GX sets the owner count of the new transform to 1. When you assign an existing transform object to a shape, QuickDraw GX increments the transform's owner count, corresponding to the new reference to the transform contained in the shape object.

For example, in Listing 6-1 on page 6-16, the call to `GXNewTransform` to create the transform `myTransform` sets its owner count to 1; the subsequent call to `GXSetShapeTransform` increments the owner count of `myTransform`, so it is 2. The call to `GXDisposeTransform` decrements the owner count of `myTransform`, making it 1 again. The transform is not deleted, which is appropriate because it is still used by the shape. If you were to call `GXSetShapeTransform` again to associate a different transform object with the shape, or call `GXDisposeShape` when the shape is no longer needed, the owner count of `myTransform` would decrement again, this time to 0, and it would be deleted.

As another example, the code in Listing 6-2 on page 6-17 clones a transform object before removing its reference from a shape. The cloning increments the transform's owner count, to ensure that the transform is not deleted when its owner count is decremented by the call to `GXSetShapeTransform` that removes it from the shape.

If you want to manage a transform's owner count directly, or if you want to know whether a transform object is shared, you can use the `GXGetTransformOwners` function to determine the owner count of a transform, and the `GXCloneTransform` and `GXDisposeTransform` functions to change the owner count of a transform. The `GXCloneTransform` function increments the transform's owner count, and the `GXDisposeTransform` function decrements the transform's owner count, freeing the memory used by the transform if the owner count goes to 0.

In the chapter “Style Objects” in this book, the section on manipulating a style object’s owner count discusses two common owner-count problems and how to avoid them. The problems are discussed in terms of style objects, but they apply equally well to transform objects. Refer to that discussion if you find that transform objects you create have owner counts that are higher or lower than you expect.

The `GXGetTransformOwners` function is described on page 6-39.

Getting and Setting a Transform Object’s Tag References

You can examine the list of references to tag objects currently associated with a transform object using the `GXGetTransformTags` function. Once you create a tag object, you can attach it to a transform object using the `GXSetTransformTags` function. You can attach as many tag objects as you like to a transform object.

Tag objects and the basic functions for manipulating them are described in the chapter “Tag Objects” in this book. That chapter also lists the common tag types defined and reserved by Apple Computer, Inc.

The `GXGetTransformTags` function is described on page 6-40. The `GXSetTransformTags` function is described on page 6-41.

Resetting Default Transform Properties

If you explicitly create a new transform with the `GXNewTransform` function and then modify its properties, or if you indirectly modify the properties of a shared transform (by calling, for example, `GXSetShapeMapping`) and thereby cause QuickDraw GX to create a new transform, that new transform has nondefault properties. If you want to restore the default transform properties, you can call the `GXResetTransform` function. This function resets the transform’s clip, mapping, view port list, and hit-test parameters to their default values, but does not alter its owner count or tag list.

The `GXResetTransform` function is described on page 6-38.

Getting, Setting, and Modifying the Transform Clip

The clip shape that you specify in a transform object controls the clipping of shapes associated with that transform. The transform clip must be a primitive shape; primitive shapes are described in the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*. QuickDraw GX provides a pair of functions (`GXGetTransformClip`, `GXSetTransformClip`) that get and set the clip of a specified transform, and another pair (`GXGetShapeClip`, `GXSetShapeClip`) that get and set the clip of the transform associated with a specified shape.

QuickDraw GX also provides another set of functions with which you can easily modify a clip shape using constructive geometry. Table 6-2 shows the constructive geometry operations you can perform between a transform clip and another shape, in order to modify the clip shape.

Table 6-2 Constructive geometry operations between transform clips and other shapes








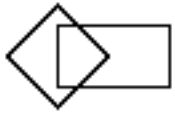






Function	Description
GXUnionTransform	Modifies the clip shape to be the union of it with another shape. Described on page 6-49.
GXIntersectTransform	Modifies the clip shape by intersecting it with another shape. Described on page 6-50.
GXDifferenceTransform	Modifies the clip shape by subtracting another shape from it. Described on page 6-51.
GXReverseDifferenceTransform	Modifies the clip shape by subtracting it from another shape. Described on page 6-52.
GXExcludeTransform	Modifies the clip shape by combining it with another shape in an exclusive-OR (XOR) operation. Described on page 6-53.

To use constructive geometry operations, the clip shape and the shape with which to operate must meet these criteria:

- n The clip shape must be a primitive shape and cannot be a picture shape, text shape, or layout shape. (These criteria are automatically met if it is a clip shape.)
- n The shape with which to operate cannot be a bitmap shape or picture shape. It should also be a primitive shape, because only its geometry and fill properties are used in the operation.
- n If the clip shape's fill is even-odd fill or winding fill, or the inverse of these, the shape with which to operate must also be filled.
- n If the clip shape is frame filled, a pen width of 0 is implied, indicating a hairline width to the clip frame. Hairlines are described in the geometric styles chapter of *Inside Macintosh: QuickDraw GX Graphics*.

Figure 6-8 shows several examples of the effects of these operations with a polygon clip combined with a rectangle shape. The figure also shows which combinations of fill types are allowed for each operation.

Figure 6-8 Constructive geometry operations with a polygon clip and a rectangle shape

	Filled clip	Filled shape	Framed clip	Filled shape	Framed clip	Framed shape
						
<code>GXUnionTransform</code>			Error			
<code>GXIntersectionTransform</code>					Error	
<code>GXDifferenceTransform</code>					Error	
<code>GXReverseDifferenceTransform</code>			Error		Error	
<code>GXExcludeTransform</code>			Error		Error	

Note

Figure 6-8 does not show a filled clip with a framed shape because this combination of shapes generates an error for any constructive geometry operation. u

The following example shows how to create a clip using a constructive geometry operation. The clip is first created as a path shape and assigned to the transform object with `GXSetTransformClip`. That clip is then unioned with another path shape, using `GXUnionTransform`. The geometries of the paths (`path1Geometry` and `path2Geometry`) are not shown.

Transform Objects

```

gxShape clipShape, pathShape;
gxTransform myTransform;
.
.  /* get or create the transform (not shown) */
.
clipShape = GXNewPaths ((gxPaths *)path1Geometry);
GXSetTransformClip(myTransform, clipShape);
GXDisposeShape(clipShape);

pathShape = GXNewPaths ((gxPaths *)path2Geometry);
GXUnionTransform(myTransform, pathShape);
GXDisposeShape(pathShape);

```

Note that only the geometries of the two path shapes matter; style information is not considered. The `GXGetTransformClip` function is described on page 6-43. The `GXSetTransformClip` function is described on page 6-44.

Moving, Scaling, Rotating, and Skewing Shapes

The mapping property of transform objects allows you to perform sophisticated transformations to your shape's geometries. By altering the values of a transform's mapping, you can move, scale, rotate, skew and create perspective effects on any shapes the transform applies to. However, determining the specific changes to the mapping matrix needed to achieve a desired transformational effect can involve complex calculations. As a convenience, QuickDraw GX provides several functions that perform the calculations necessary to achieve common transformations, without you having to know how the mapping matrix is altered.

The transformational functions that QuickDraw GX provides allow you to position, rotate, scale, and skew shapes. QuickDraw GX provides two kinds of such functions, one kind that operates on transform mappings, which is of the form `GXActionTransform`, and one kind that normally operates on shape geometries, which is of the form `GXActionShape`. If you use a function that operates on a transform's mapping, the mapping is changed and all shapes that refer to the transform are affected. If you use a function that normally operates on a shape geometry, there are two possible results:

- n If the shape's `gxMapTransformShape` attribute is cleared, the shape's geometry is changed, as expected. Its transform mapping is unaffected.
- n If the shape's `gxMapTransformShape` attribute is set, the function works exactly like a `GXActionTransform` function, changing the transform mapping instead of the shape geometry. An additional side effect is that, if the shape's transform object is shared with other shapes, QuickDraw GX creates a copy of the transform and modifies the copy, to avoid affecting the other shapes.

Transform Objects

If you move a shape to an absolute location, the location applies to a specific anchor point in the shape's geometry and all other points in the geometry move in relation to this point. The point used depends on the kind of shape:

- n For points, lines, and curves, the anchor point is the first point in the shape's geometry.
- n For rectangles, polygons, paths, and bitmaps, the anchor point is the top-left corner of the bounding rectangle.
- n For text, glyph, and layout shapes, the anchor point is the origin of the first glyph.
- n Other shapes (empty shapes, full shapes, and pictures) cannot be moved.

However, remember that if the shape's `gxMapTransformShape` attribute is set, calling a function that moves the shape has no effect on the geometry; it modifies the transform mapping instead. In that case, moving the shape to an absolute location means only that its transform mapping adds that location to whatever location the geometry already specifies.

Modifying the Transform Mapping

One way to transform a shape is by altering its transform object's mapping property. This section shows several examples of that kind of transformation.

For example, you can move a shape to a relative or absolute location by modifying its transform. The `GXMoveTransform` function modifies the transform's mapping to move a shape a specified distance from its current location in local coordinates. The `GXMoveTransformTo` function modifies the transform's mapping to move a shape to an absolute location in local coordinate space.

The following example causes all shapes associated with the `myTransform` transform object to move to the upper-left corner of the bounding rectangle, `bounds`, of the rectangle `aRectangle`:

```
gxRectangle aRectangle, bounds;
.
. /* initialize the rectangle (not shown) */
.
GXGetShapeBounds(aRectangle, 0, &bounds);
GXMoveTransformTo(myTransform, bounds.left, bounds.top);
```

You can also modify a transform's mapping to rotate, scale, or skew a shape around a specified point. Listing 6-3 rotates a shape's transform mapping 90 degrees, and scales and skews the mapping. The shape's center is used as the point around which to rotate, scale, and skew the transform.

Listing 6-3 Modifying a shape's transform with transform-mapping calls only

```

Fixed          hScale, vScale, xSkew, ySkew;
gxPoint       center;
gxShape       aShape;
gxTransform   myTransform;
.
.  /* initialize the shape and the rotate/scale/skew parameters */
.
/* find the shape's center */
GXGetShapeCenter(aShape, 0, &center);

/* get the transform, rotate it around shape's center */
myTransform = GXGetShapeTransform(aShape);
GXRotateTransform(myTransform, ff(90), center.x, center.y);

/* scale and skew the shape */
GXScaleTransform(myTransform, hScale, vScale, center.x, center.y);
GXSkewTransform(myTransform, xSkew, ySkew, center.x, center.y);

```

Listing 6-4 performs the same actions as Listing 6-3: rotating, scaling, and skewing a shape's transform mapping. Like Listing 6-3, this code also affects only the transform mapping associated with the shape. This is despite the fact that it makes some calls (GXScaleShape and GXSkewShape) that would normally affect the shape's geometry. Because the shape's gxMapTransformShape attribute is set before the geometry-altering calls are made, those functions are forced to affect the transform mapping instead of the shape's geometry.

Listing 6-4 Modifying a shape's transform with transform-mapping and shape-geometry calls

```

Fixed          hScale, vScale, xSkew, ySkew;
gxPoint       center;
gxShape       aShape;
gxTransform   myTransform;
.
.  /* initialize the shape and the rotate/scale/skew parameters */
.
/* find the shape's center */
GXGetShapeCenter(aShape, 0, &center);

/* get the transform, rotate it around shape's center */
myTransform = GXGetShapeTransform(aShape);
GXRotateTransform(myTransform, ff(90), center.x, center.y);

```

Transform Objects

```

/* set the gxMapTransformShape attribute */
GXSetShapeAttributes(aShape,
                    GXGetShapeAttributes(aShape) | gxMapTransformShape);

/* scale and skew the shape (but it affects mapping instead) */
GXScaleShape(aShape, hScale, vScale, center.x, center.y);
GXSkewShape(aShape, xSkew, ySkew, center.x, center.y);

```

You can also perform these transformations, as well as perspective-modifying operations, by directly manipulating the matrix elements of a transform's mapping. You can use the functions `GXGetTransformMapping` or `GXGetShapeMapping` to obtain the mapping matrix, then modify the matrix as desired and reassign it to the transform with `GXSetTransformMapping` or `GXSetShapeMapping`. You can also create your own mapping matrix, and then multiply it (concatenate it) with the existing mapping of a transform object, using the functions `GXMapTransform` (or `GXMapShape`, if the shape's `gxMapTransformShape` attribute is set). For more information about matrix manipulation, see the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

The `GXGetTransformMapping` function is described on page 6-54; the `GXSetTransformMapping` function is described on page 6-55.

The `GXGetShapeMapping` function is described on page 6-56; the `GXSetShapeMapping` function is described on page 6-57.

The `GXMapTransform` function is described on page 6-64; the `GXMapShape` function is described on page 6-72.

The `GXMoveTransform` function is described on page 6-58. The `GXMoveTransformTo` function is described on page 6-59. The `GXScaleTransform` function is described on page 6-60. The `GXRotateTransform` function is described on page 6-62. The `GXSkewTransform` function is described on page 6-63.

Modifying Shape Geometry

A second way to transform a shape is by altering its geometry property. This section shows several examples of that kind of transformation.

You can move a shape to a relative or absolute location by modifying the shape's geometry instead of its transform mapping. The `GXMoveShape` function modifies the geometry to move a shape a specified distance from its current location in local coordinates. The `GXMoveShapeTo` function modifies the geometry to move a shape to an absolute location in local coordinate space. In either case, the geometry is altered only if the shape's `gxMapTransformShape` attribute is cleared; otherwise, the functions work just like `GXMoveTransform` and `GXMoveTransformTo`, and alter the mapping of the transform object attached to the shape.

Transform Objects

The following example causes the shape `myShape` to move to the upper-left corner of the bounding rectangle, `bounds`, of the rectangle `aRectangle`:

```
gxRectangle aRectangle, bounds;
.
. /* initialize the rectangle (not shown) */
.
GXGetShapeBounds(aRectangle, 0, &bounds);
GXMoveShapeTo(myShape, bounds.left, bounds.top);
```

Listing 6-5 performs the same actions as Listing 6-3 and Listing 6-4: rotating, scaling, and skewing a shape. However, unlike either previous listing, this code alters the geometry of the shape itself. To ensure that the operations do not affect the shape's transform mapping, the code clears the shape's `gxMapTransformShape` attribute before making the geometry-altering calls.

Listing 6-5 Modifying a shape's geometry with shape-geometry calls

```
Fixed          hScale, vScale, xSkew, ySkew;
gxPoint        center;
gxShape        aShape;
.
. /* initialize the shape and the rotate/scale/skew parameters */
.
/* find the shape's center */
GXGetShapeCenter(aShape, 0, &center);

/* clear the gxMapTransformShape attribute */
GXSetShapeAttributes(aShape, gxNoAttributes);

/* rotate shape around its center (affects geometry this time) */
GXRotateShape(myShape, ff(90), center.x, center.y);

/* scale and skew the shape (affects geometry this time) */
GXScaleShape(aShape, hScale, vScale, center.x, center.y);
GXSkewShape(aShape, xSkew, ySkew, center.x, center.y);
```

Note

Rotation of a shape's geometry can change the shape's type. For example, a rectangle may turn into a polygon when rotated. For more information, see the geometric shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*. u

You can also perform these operations, as well as perspective-modifying operations, by applying a mapping directly to a shape's geometry. You can create your own mapping matrix, and then apply it to a shape object using the `GXMapShape` function if the shape's `gxMapTransformShape` attribute is cleared; if the attribute is set, this function affects the shape's transform mapping instead. For more information about matrix manipulation, see the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

The `GXMapShape` function is described on page 6-72.

The `GXMoveShape` function is described on page 6-66. The `GXMoveShapeTo` function is described on page 6-67. The `GXScaleShape` function is described on page 6-68. The `GXRotateShape` function is described on page 6-70. The `GXSkewShape` function is described on page 6-71.

Manipulating the View Port List

The view port list property of the transform object specifies all the view ports to which shapes associated with the transform are drawn. QuickDraw GX provides a pair of functions (`GXGetTransformViewPorts` and `GXSetTransformViewPorts`) that get and set the view port list of a specified transform, and another pair (`GXGetShapeViewPorts` and `GXSetShapeViewPorts`) that get and set the view port list of the transform associated with a specified shape. View ports are described in the chapter "View-Related Objects" in this book.

When you create a window, you create one or more view ports. If you want the shapes that you subsequently create to be drawn in that window, you place references to one or more of those view ports in the view port list of the shapes' transform object.

You may also want to alter a view port list of an existing transform object. For example, you might temporarily create a pane or a separate window that shows a zoomed view of a currently displayed shape. As another example, you might want to draw an object both onscreen and offscreen simultaneously.

Listing 6-6 is a partial listing of a function that adds a new view port (`newPort`) to the view port list of the transform object `myTransform`. The function calls the `GXGetTransformViewPorts` function twice, first to determine the number of view ports already in the list in order to allocate memory for it, and second to retrieve the list itself. Before adding the new view port, the function first checks the list and, if the view port is already in the list, does not add it. The function assigns the new view port to the last element in the list, and then calls `GXSetTransformViewPorts` to reassign the list to the transform. Finally, the code disposes of the view port list.

Listing 6-6 Getting and setting view ports

```

gxViewPort    *ports, *portPtr;
gxViewPort    newPort;
short         portCount, count;
gxTransform   myTransform;
.
.  /* get the transform, set up the new view port (not shown) */
.
/* first, call to see how big the current view port list is */
portCount = GXGetTransformViewPorts(myTransform, nil);

/* accounting for new view port, allocate memory for the list */
portCount++;
ports = (gxViewPort *) NewPtr(portCount * sizeof(gxViewPort));

/* get the current list into memory */
GXGetTransformViewPorts(myTransform, ports);

/* check if the view port is already in the list */
portPtr = ports;
count = portCount;
while (--count > 0)
{
    /* if port is already in transform, release memory and leave */
    if (*portPtr++ == newPort)
    {
        DisposPtr((void *) ports);
        return;
    }
}
/* put view port in transform */
*portPtr = newPort;
GXSetTransformViewPorts(myTransform, portCount, ports);

/* clean up and leave */
DisposePtr((void *)ports);
return;

```

The `GXGetTransformViewPorts` function is described on page 6-73;
the `GXSetTransformViewPorts` function is described on page 6-74. The
`GXGetShapeViewPorts` function is described on page 6-75;
the `GXSetShapeViewPorts` function is described on page 6-76.

Setting Up Hit-Test Parameters

QuickDraw GX provides a pair of functions (`GXGetTransformHitTest`, `GXSetTransformHitTest`) that get and set the hit-test parameters of a specified transform, and another pair (`GXGetShapeHitTest`, `GXSetShapeHitTest`) that get and set the hit-test parameters of the transform associated with a specified shape.

The hit-test parameters are used by the functions `GXHitTestShape` and `GXHitTestPicture`. Before calling either function, you set up the shape-parts mask and define a tolerance, and assign them both to the transform object of the shape you are going to hit-test. The shape-parts mask consists of values from the `gxShapeParts` enumeration; see Table 6-1 on page 6-12 for descriptions of the individual values.

The `GXHitTestShape` and `GXHitTestPicture` functions return, in addition to an indication of which shape parts were hit during a hit-test, a distance from the hit point to one of the hit parts. If only one shape part was hit, the distance is the distance from the hit point to the nearest point on the hit part. But if more than one part was hit (for example, if a hit corresponded to both the bounding rectangle and the shape geometry), the distance returned is the distance to the first shape part—in order of processing by the function—that was hit. The order in which shape parts are processed is the order in which they appear in the `gxShapeParts` enumeration. Thus, if both bounding rectangle and geometry are tested for, and if both are hit, the distance returned is the distance to the bounding rectangle. You can use the processing order to set up the shape-parts mask to make sure that `GXHitTestShape` and `GXHitTestPicture` return the exact distance information you need.

The following example sets the shape-parts mask (`mask`) to include both the geometry and the corner points of the shape `aShape`. It also sets the tolerance to 0, meaning that if a hit point is any distance outside of the shape geometry or corner points, it is not considered a hit.

```
gxShapePart    mask = gxGeometryPart | gxCornerPointPart;  
GXSetShapeHitTest(aShape, mask, 0);
```


For more information about shape parts and tolerance as a transform object property, see the section “Hit-Test Parameters” on page 6-11. For information about hit-testing with `GXHitTestShape`, see the chapter “Shape Objects” in this book. For information about hit-testing with `GXHitTestPicture`, see the picture shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*.

The `GXGetTransformHitTest` function is described on page 6-78; the `GXSetTransformHitTest` function is described on page 6-79. The `GXGetShapeHitTest` function is described on page 6-80; the `GXSetShapeHitTest` function is described on page 6-81.

Transform Objects Reference

This section provides reference information to the constants, data types, and functions that allow you to create and manipulate transform objects and alter their properties. It includes

- n definitions of the constants and data types, including enumerations, that are specific to transform objects
- n descriptions of the QuickDraw GX functions that operate on transform objects
- n descriptions of the QuickDraw GX transformation functions that operate either on shape geometries or on transform mappings, depending on the state of the `gxMapTransformShape` attribute

Constants and Data Types

This section describes the data types that you use to obtain and provide information about transform objects.

The Transform Object

QuickDraw GX provides you with access to an individual transform object through a transform reference:

```
typedef struct gxPrivateTransformRecord *gxTransform;
```

In this type definition, `gxTransform` is a type-checked reference, not an actual pointer to any defined structure. The contents of the transform object are private.

Shape Parts for Hit-Testing

Each transform object specifies the parts of a shape on which hit-testing is performed. The choices are specified by the `gxShapeParts` enumeration. For determining distance to a hit part, the hit-testing functions evaluate shape parts in the order shown in the enumeration.

```
enum gxShapeParts {          /* (in order of evaluation) */
    gxNoPart                 = 0,
    gxBoundsPart             = 0x0001,
    gxGeometryPart          = 0x0002,
    gxPenPart               = 0x0004,
    gxCornerPointPart       = 0x0008,
    gxControlPointPart      = 0x0010,
    gxEdgePart              = 0x0020,
    gxJoinPart              = 0x0040,
    gxStartCapPart          = 0x0080,
    gxEndCapPart            = 0x0100,
    gxDashPart              = 0x0200,
    gxPatternPart           = 0x0400,
    gxGlyphBoundsPart       = gxJoinPart,
    gxGlyphFirstPart        = gxStartCapPart,
    gxGlyphLastPart         = gxEndCapPart,
    gxSideBearingPart       = gxDashPart,
    gxAnyPart                = gxBoundsPart | gxGeometryPart |
        gxPenPart | gxCornerPointPart | gxControlPointPart |
        gxEdgePart | gxJoinPart | gxStartCapPart |
        gxEndCapPart | gxDashPart | gxPatternPart
};

typedef long gxShapePart;
```

The individual shape parts are described in Table 6-1 on page 6-12.

Functions

This section describes the QuickDraw GX functions you can use to

- n create and manipulate a transform object
- n manipulate the common object properties of a transform object
- n get and set the clip shape of a transform object
- n perform geometric operations on a transform clip
- n get and set the mapping matrix of a transform object

Transform Objects

- n apply transformation operations to a transform's mapping
- n apply transformation operations directly to a shape's geometry
- n get and set the view port list of a transform object
- n get and set the hit-test parameters of a transform object

Creating and Manipulating Transform Objects

This section describes the functions that manipulate transforms as objects in memory. With the functions in this section, you can create and dispose of a transform object, and copy, compare, and clone transform objects.

To associate a transform object with a QuickDraw GX shape, use the `GXGetShapeTransform` and `GXSetShapeTransform` functions, described in the chapter "Shape Objects" in this book.

GXNewTransform

You can use the `GXNewTransform` function to create a new transform object with default properties.

```
gxTransform GXNewTransform(void);
```

function result A reference to the newly created transform object.

DESCRIPTION

The `GXNewTransform` function creates a transform object with an owner count of 1. All other properties of the transform are set to their default values:

- n A clip that is a full shape.
- n A mapping that is the identity mapping.
- n A shape-parts mask specifying `gxBoundsPart` only, and a tolerance of 0.
- n A view port list containing a single view port covering all onscreen view devices.
- n The owner count is 1.
- n The tag list is empty.

SPECIAL CONSIDERATIONS

If no error occurs, the `GXNewTransform` function creates a transform object; you are responsible for disposing of that object when you no longer need it.

ERRORS, WARNINGS, AND NOTICES**Errors**`out_of_memory`**SEE ALSO**

For an example of the use of this function, see Listing 6-1 on page 6-16.

Default transform properties are described in the section “Default Transform Objects” beginning on page 6-14. For general information on the properties of transform objects, see “Transform Object Properties” beginning on page 6-6.

To dispose of a transform object, use the `GXDisposeTransform` function, which is described in the next section.

To create a transform object that is identical to an existing one, use the `GXCopyToTransform` function, described on page 6-35.

GXDisposeTransform

You can use the `GXDisposeTransform` function to release a reference to a transform object.

```
void GXDisposeTransform(gxTransform target);
```

`target` The transform object to dispose of.

DESCRIPTION

The `GXDisposeTransform` function decrements the owner count of the transform object specified by the `target` parameter, and releases any memory used by the transform if the owner count goes to 0.

ERRORS, WARNINGS, AND NOTICES**Errors**`transform_is_nil`**Warnings**`cannot_dispose_default_transform` (debugging version)

SEE ALSO

For an example of the use of this function, see Listing 6-1 on page 6-16.

Owner counts for transform objects are discussed in the section “Copying, Comparing, and Cloning Transform Objects” beginning on page 6-16, and in the section “Manipulating a Transform Object’s Owner Count” beginning on page 6-19.

To examine the owner count of a transform, use the `GXGetTransformOwners` function, described on page 6-39. To increment the owner count of a transform object, use the `GXCloneTransform` function, which is described on page 6-37.

GXCopyToTransform

You can use the `GXCopyToTransform` function to create a copy of an existing transform object.

```
gxTransform GXCopyToTransform (gxTransform target,
                               gxTransform source);
```

target A reference to the transform object to copy the source contents into. If you specify `nil` for this parameter, the `GXCopyToTransform` function creates a new transform object.

source A reference to the transform object whose contents you want to copy.

DESCRIPTION

The `GXCopyToTransform` function copies the contents of an existing transform object to another, or it creates a new transform object and copies the contents of an existing transform object to it. The function copies the clip, mapping, hit-test parameters, tag list and view port list (but not the owner count) of the source transform object into the target transform object. It clones, but does not copy, the tag objects in the tag list.

If you specify `nil` for the `target` parameter, the `GXCopyToTransform` function creates a new transform object and copies the properties of the source transform, including the tag list, to the new transform.

You can use this function to create a copy of a transform object and then modify it without changing the original.

SPECIAL CONSIDERATIONS

If you specify `nil` for the `target` parameter and no error occurs, the `GXCopyToTransform` function creates a transform object; you are responsible for disposing of that object when you no longer need it.

ERRORS, WARNINGS, AND NOTICES

Errors

out_of_memory
transform_is_nil

SEE ALSO

For an example of the use of this function, see page 6-17.

To create a new transform object with default values, use the `GXNewTransform` function, described on page 6-33.

To compare transform objects for equality, use the `GXEqualTransform` function, described in the next section.

GXEqualTransform

You can use the `GXEqualTransform` function to determine if two transform objects are equal.

```
boolean GXEqualTransform(gxTransform one, gxTransform two);
```

`one` A reference to one of the transform objects to test for equality.

`two` A reference to the other transform object to test for equality.

function result `true` if the transform specified by the `one` parameter is equal to the transform specified by the `two` parameter; otherwise `false`.

DESCRIPTION

The `GXEqualTransform` function determines whether the transform object referenced by the `one` parameter is equal to the transform object referenced by the `two` parameter.

For two transform objects to be equal, they must have identical clip shape geometries, mappings, hit-test parameters, and view port lists. Their owner count and tag list need not be identical.

SPECIAL CONSIDERATIONS

Note that for two clips to be identical means more than having identical dimensions. For example, a polygon clip might have the same dimensions as a path or rectangle, but shapes with different shape types are never identical. You can call the `GXSimplifyShape` function to convert the clips to their simplest form.

ERRORS, WARNINGS, AND NOTICES**Errors**

out_of_memory
transform_is_nil

SEE ALSO

To make a copy of a transform object that is equal by the criteria of this function, use the `GXCopyToTransform` function, described on page 6-35.

The `GXSimplifyShape` function is described in the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*.

GXCloneTransform

You can use the `GXCloneTransform` function to clone a transform object—that is, to add a reference to it and increment its owner count.

```
gxTransform GXCloneTransform(gxTransform source);
```

`source` A reference to the transform object to clone.

function result A copy of the reference to the source transform object.

DESCRIPTION

The `GXCloneTransform` function increments the owner count of the transform referenced by the `source` parameter. You typically use this function when you want to create another reference to an existing transform rather than creating a distinct copy of the transform.

This function returns as its function result a reference to the transform—the same reference you pass in as the `source` parameter. It also increments the transform's owner count.

ERRORS, WARNINGS, AND NOTICES**Errors**

transform_is_nil

SEE ALSO

For an example of the use of this function, see Listing 6-2 on page 6-17.

Owner counts for transform objects are discussed in the section “Copying, Comparing, and Cloning Transform Objects” beginning on page 6-16, and in the section “Manipulating a Transform Object’s Owner Count” beginning on page 6-19.

To examine the owner count of a transform, use the `GXGetTransformOwners` function, described on page 6-39. To decrement the owner count of a transform, use the `GXDisposeTransform` function, described on page 6-34.

Manipulating Transform Object Properties

This section describes the functions that manipulate the object properties of transforms. The functions described in this section allow you to

- n reset a transform object’s properties to their default values
- n manipulate the common object properties of a transform: owner count and tag list

GXResetTransform

You can use the `GXResetTransform` function to reset the properties of a transform object to their default values.

```
void GXResetTransform(gxTransform target);
```

`target` A reference to the transform object whose properties you want to reset.

DESCRIPTION

The `GXResetTransform` function resets the following properties of the target transform to the following default values:

- n The clip is a full shape.
- n The mapping is the identity mapping.
- n The shape-parts mask specifies `gxBoundsPart`, and the tolerance is 0.
- n The view port list contains a single view port covering all screen view devices.

This function does not change the target transform’s owner count or tag list.

ERRORS, WARNINGS, AND NOTICES**Errors**

out_of_memory
transform_is_nil

SEE ALSO

Default transform properties are described in the section “Default Transform Objects” beginning on page 6-14. For general information on the properties of transform objects, see “Transform Object Properties” beginning on page 6-6.

GXGetTransformOwners

You can use the `GXGetTransformOwners` function to determine the number of references to a particular transform object.

```
long GXGetTransformOwners(gxTransform source);
```

`source` A reference to the transform object whose owner count you want to find.

function result The owner count of the source transform object.

DESCRIPTION

The `GXGetTransformOwners` function returns a value indicating the number of current references to the source shape.

ERRORS, WARNINGS, AND NOTICES**Errors**

transform_is_nil

SEE ALSO

Owner counts for transform objects are discussed in the section “Copying, Comparing, and Cloning Transform Objects” beginning on page 6-16, and in the section “Manipulating a Transform Object’s Owner Count” beginning on page 6-19.

To increment the owner count of a transform object, use the `GXCloneTransform` function, described on page 6-37. To release a reference to a transform object, use the `GXDisposeTransform` function, described on page 6-34.

GXGetTransformTags

You can use the `GXGetTransformTags` function to examine one or more of the tag objects associated with a transform object.

```
long GXGetTransformTags(gxTransform source, long tagType,
                        long index, long count, gxTag items[]);
```

<code>source</code>	A reference to the transform object whose tag list you want to examine.
<code>tagType</code>	The type of tag object to search for. A value of 0 indicates that you want to look for all tag types.
<code>index</code>	The (1-based) index of the first such tag reference to return.
<code>count</code>	The number of tag references to return.
<code>items</code>	An array to hold the returned tag references.

function result The number of tag references found that match the criteria specified by the input parameters.

DESCRIPTION

The `GXGetTransformTags` function searches the tag list of the source transform object for references to tag objects with the tag type specified by the `tagType` parameter. If you specify 0 for this parameter, the function searches for all tag types.

You can use the `index` and the `count` parameters to specify which tag references of the appropriate type the `GXGetInkTags` function should return. The `index` parameter indicates the first tag reference to return and the `count` parameter indicates how many tag references to return. The `index` parameter must be greater than 0. The `count` parameter must be greater than 0 or equal to the `gxSelectToEnd` constant (-1), which indicates that all tag references (starting with the tag reference indicated by the `index` parameter) should be returned.

The function result is the number of tag references found that fit the criteria. If you pass a value other than `nil` for the `items` parameter, the `GXGetInkTags` function returns in the `items` parameter the tag references that were found.

Typically, you call this function once with a `nil` value for the `items` parameter to determine the number of matching tags. Then you allocate an appropriately sized tag reference array and call the function a second time to obtain references to the matching tags.

ERRORS, WARNINGS, AND NOTICES**Errors**

```

out_of_memory
transform_is_nil
index_is_less_than_one    (debugging version)
count_is_less_than_one    (debugging version)

```

Warnings

```

index_out_of_range
count_out_of_range

```

SEE ALSO

Tag objects are discussed in the chapter “Tag Objects” in this book.

To change the set of tags associated with a transform, use the `GXSetTransformTags` function, described next.

GXSetTransformTags

You can use the `GXSetTransformTags` function to add, remove, or replace tag objects associated with a transform object.

```

void GXSetTransformTags(gxTransform target, long tagType,
                        long index, long oldCount, long newCount,
                        const gxTag items[]);

```

<code>target</code>	A reference to the transform object whose tag list you want to alter.
<code>tagType</code>	The type of tag objects to replace. A value of 0 indicates that you want to replace tags of all types.
<code>index</code>	The (1-based) index of the first tag reference (of the appropriate type) to replace.
<code>oldCount</code>	The number of tag references to replace. A value of 0 specifies that you want to insert tag references before the tag reference indicated by the <code>index</code> parameter, rather than replace tag references. A value of -1 (the <code>gxSelectToEnd</code> constant) indicates that all tag references of the requested type (starting with the reference indicated by the <code>index</code> parameter), should be replaced.
<code>newCount</code>	The number of tag references to insert. A value of 0 indicates that there are no tag references to insert; the existing tag references that match the selection criteria are removed from the target transform’s tag list and disposed of.
<code>items</code>	An array of tag references to insert into the transform’s tag list.

DESCRIPTION

The `GXSetTransformTags` function allows you add tag references to a transform object's tag list, to remove tag references from the list, or to replace tag references in the list with new tag references. In any of these three cases, the `target` parameter specifies the ink object to be modified, the `newCount` parameter specifies the number of tag references to add, and the `items` parameter provides the new tag references.

- n To add tag references, set the `oldCount` parameter to 0. Use the `tagType` and the `index` parameters to specify where to add the new tag references. (For example, if you specify `nil` for the `tagType` parameter and 1 for the `index` parameter, this function inserts the new tag references before the current tag references. If you specify a value other than `nil` for the `tagType` parameter and a value of 2 for the `index` parameter, the function inserts the new tag references before the second tag reference with a tag type matching the `tagType` parameter.)
- n To remove tag references, set the `newCount` parameter to 0 and the `items` parameter to `nil`. You can use the `index` and the `oldCount` parameters to specify which tag references (of the specified type) should be removed. The `index` parameter indicates the first tag reference (of the specified type) to remove and the `oldCount` parameter indicates how many tag references (of the specified type) to remove.
- n To replace tag references, use the `tagType`, `index`, and `oldCount` parameters to indicate which tag references to replace, and use the `newCount` and `items` parameters to specify the new tag references to add. If `newCount` is greater than `oldCount`, the extra tag references are placed immediately adjacent to the last tag reference replaced.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>out_of_memory</code>	
<code>transform_is_nil</code>	
<code>tag_is_nil</code>	
<code>parameter_is_nil</code>	(debugging version)
<code>inconsistent_parameters</code>	(debugging version)
<code>parameter_out_of_range</code>	(debugging version)
<code>index_is_less_than_zero</code>	(debugging version)
<code>cannot_dispose_locked_tag</code>	(debugging version)

Warnings

`index_out_of_range`
`count_out_of_range`

Notices (debugging version)

`tag_already_set`

SEE ALSO

Tag objects are discussed in the chapter “Tag Objects” in this book.

To examine the set of tags associated with a transform, use the `GXGetTransformTags` function, described in the previous section.

Getting and Setting the Clip

This section describes the functions that allow you to manipulate the clip property of a transform object. The clip property is a reference to a shape object. See “Clip” on page 6-7 for more information.

GXGetTransformClip

You can use the `GXGetTransformClip` function to retrieve the clip property of a transform object.

```
gxShape GXGetTransformClip(gxTransform source);
```

source A reference to the transform object whose clip shape you want to determine.

function result A reference to a newly created shape object that is a copy of the source transform’s clip.

DESCRIPTION

The `GXGetTransformClip` function creates a new shape object, copies into it the geometry of the shape referenced in the clip property of the source transform, and returns a reference to the new shape as the function result.

Note that the returned shape object is a copy; you can alter it without affecting the clip property of the source transform. If you call this function and alter the clip shape it returns, you can then assign that changed clip shape back to the transform object by calling the `GXSetTransformClip` function.

SPECIAL CONSIDERATIONS

If no error occurs, the `GXGetTransformClip` function creates a shape object; you are responsible for disposing of that object when you no longer need it.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
transform_is_nil
parameter_is_nil      (debugging version)
```

SEE ALSO

For information about the clip property of transform objects, see “Clip” on page 6-7.

To change the information in the clip property of a transform object, use the `GXSetTransformClip` function, described next.

If you want to manipulate the clip property of a transform associated with a specific shape, you can use the `GXGetShapeClip` function, described on page 6-45, or the `GXSetShapeClip` function, described on page 6-46.

GXSetTransformClip

You can use the `GXSetTransformClip` function to change a transform object’s clip property.

```
void GXSetTransformClip(gxTransform target, gxShape clip);
```

`target` A reference to the transform object whose clip shape you want to change.
`clip` A reference to a shape object containing the new clip shape information.

DESCRIPTION

The `GXSetTransformClip` function copies information from the shape object referenced by the `clip` parameter into the clip property of the transform object referenced by the `target` parameter. You can specify `nil` for the `clip` parameter, in which case this function sets the clip property of the target transform to a full clip (no transform clipping takes place).

The new clip shape, which you specify using the `clip` parameter, may be a geometric shape, a bitmap shape, or a glyph shape. It may not be a picture, text, or layout shape.

- n If you specify a geometric shape, it must be in primitive form—that is, all the stylistic information about the shape must be incorporated into the shape’s geometry—because this function copies only the geometry-related information from the shape you specify. It does not copy the information contained in the shape’s style. You can convert a shape to its primitive form using the `GXPrimitiveShape` function, which is described in *Inside Macintosh: QuickDraw GX Graphics*. You can also specify an empty or full shape for a clip.
- n If you specify a bitmap shape, it must have a pixel size of 1 and its color profile reference must be `nil`. In the bitmap, pixel values of 0 obscure drawing; pixel values of 1 do not restrict visibility.
- n If you specify a glyph shape, this function uses information from the glyph shape’s style object as well as its style list to determine the size, form, and position of the glyph outlines; those outlines are then used to clip drawing. The style list cannot have `nil` entries. A style object referenced by the glyph shape cannot be complex—that is, it cannot have a cap, join, dash, pattern, text face, font variation, tag list, or any of the properties used only by layout shapes.

ERRORS, WARNINGS, AND NOTICES

Errors

<code>out_of_memory</code>	
<code>transform_is_nil</code>	
<code>shape_is_nil</code>	
<code>shapeFill_not_allowed</code>	(debugging version)
<code>colorProfile_must_be_nil</code>	(debugging version)
<code>bitmap_pixel_size_must_be_1</code>	(debugging version)
<code>empty_shape_not_allowed</code>	(debugging version)
<code>ignorePlatformShape_not_allowed</code>	(debugging version)
<code>nil_style_in_glyph_not_allowed</code>	(debugging version)
<code>complex_glyph_style_not_allowed</code>	(debugging version)
<code>illegal_type_for_shape</code>	(debugging version)

Warnings

<code>tags_in_shape_ignored</code>	(debugging version)
------------------------------------	---------------------

Notices (debugging version)

<code>clip_already_set</code>	
-------------------------------	--

SEE ALSO

For an example of the use of this function, see page 6-23.

For information about the clip property of transform objects, see “Clip” on page 6-7.

For information about primitive shapes, geometric shapes and bitmap shapes, see *Inside Macintosh: QuickDraw GX Graphics*. For information about glyph shapes, see *Inside Macintosh: QuickDraw GX Typography*.

To examine the clip shape of a transform object, use the `GXGetTransformClip` function, described in the previous section.

If you want to manipulate the clip property of a transform associated with a specific shape, you can use the `GXGetShapeClip` function, described next, or the `GXSetShapeClip` function, described on page 6-46.

GXGetShapeClip

You can use the `GXGetShapeClip` function to retrieve the clip property of a transform object associated with a specified shape.

```
gxShape GXGetShapeClip(gxShape source);
```

source A reference to the shape whose transform object you want to examine the clip property of.

function result A reference to a newly created shape encapsulating information copied from the clip property of the source shape’s transform object.

DESCRIPTION

The `GXGetShapeClip` function creates a new shape object, copies into it geometry information from the clip property of the source shape's transform object, and returns a reference to the new shape as the function result.

Note that the returned shape object is a copy; you can alter it without affecting the clip property of the source shape's transform. If you call this function and alter the clip shape it returns, you can then assign that changed clip shape back to the shape's transform object by calling the `GXSetShapeClip` function.

SPECIAL CONSIDERATIONS

If no error occurs, the `GXGetShapeClip` function creates a shape object; you are responsible for disposing of that object when you no longer need it.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`shape_is_nil`

SEE ALSO

For information about the clip property of transform objects, see "Clip" on page 6-7.

To alter the clip property of a transform object associated with a particular shape, use the `GXSetShapeClip` function, described in the next section.

If you want to manipulate the clip property of a particular transform object, you can use the `GXGetTransformClip` function, described on page 6-43, or the `GXSetTransformClip` function, described on page 6-44.

GXSetShapeClip

You can use the `GXSetShapeClip` function to change the clip property of a transform object associated with a specified shape.

```
void GXSetShapeClip(gxShape target, gxShape clip);
```

`target` A reference to the shape whose transform object you want to change the clip shape of.

`clip` A reference to a shape object containing the new clip shape information.

DESCRIPTION

The `GXSetShapeClip` function copies information from the shape object referenced by the `clip` parameter into the `clip` property of the transform object associated with the shape referenced by the `target` parameter.

Calling this function is almost equivalent to

```
GXSetTransformClip(GXGetShapeTransform(myShape), theClip);
```

except that, if the source shape's transform object is shared with other shapes, `GXSetShapeClip` creates a new copy of the transform object, attaches it to the source shape, and changes the `clip` of the copy. That way, calling this function does not produce side effects on other shapes.

You can specify `nil` for the `clip` parameter, in which case this function sets the `clip` property of the target shape's transform to a full clip (no transform clipping takes place).

The new clip shape, which you specify using the `clip` parameter, may be a geometric shape, a bitmap shape, or a glyph shape. It may not be a picture, text, or layout shape.

- n If you specify a geometric shape, it must be in primitive form—that is, all the stylistic information about the shape must be incorporated into the shape's geometry—because this function copies only the geometry-related information from the shape you specify. It does not copy the information contained in the shape's style. You can convert a shape to its primitive form using the `GXPrimitiveShape` function, which is described in *Inside Macintosh: QuickDraw GX Graphics*.
- n If you specify a bitmap shape, it must have a pixel size of 1 and its color profile reference must be `nil`. In the bitmap, pixel values of 0 obscure drawing; pixel values of 1 do not restrict visibility.
- n If you specify a glyph shape, this function uses information from the glyph shape's style object as well as its style list to determine the size, form, and position of the glyph outlines; those outlines are then used to clip drawing. The style list cannot have `nil` entries. A style object referenced by the glyph shape cannot be complex—that is, it cannot have a cap, join, dash, pattern, text face, font variation, tag list, or any of the properties used only by layout shapes.

ERRORS, WARNINGS, AND NOTICES

Errors

out_of_memory	
transform_is_nil	
shape_is_nil	
shapeFill_not_allowed	(debugging version)
colorProfile_must_be_nil	(debugging version)
bitmap_pixel_size_must_be_1	(debugging version)
empty_shape_not_allowed	(debugging version)
ignorePlatformShape_not_allowed	(debugging version)
nil_style_in_glyph_not_allowed	(debugging version)
complex_glyph_style_not_allowed	(debugging version)
illegal_type_for_shape	(debugging version)

Warnings

tags_in_shape_ignored	(debugging version)
-----------------------	---------------------

Notices (debugging version)

clip_already_set	
------------------	--

SEE ALSO

To retrieve the clip property of a transform object associated with a particular shape, use the `GXGetShapeClip` function, described in the previous section.

To assign a clip directly to a transform object, use the `GXSetTransformClip` function, described on page 6-44.

For information about the clip property of transform objects, see “Clip” on page 6-7.

For information about primitive shapes, geometric shapes and bitmap shapes, see *Inside Macintosh: QuickDraw GX Graphics*. For information about glyph shapes, see *Inside Macintosh: QuickDraw GX Typography*.

Performing Geometric Operations on Transform Clips

QuickDraw GX provides a number of functions that allow you to perform constructive geometry operations on the clip shapes of transform objects. Each of these functions replaces the clip property of a transform object with the result of an operation combining the original clip geometry with the geometry of another shape. The functions are

- n `GXUnionTransform`, which combines the transform clip with a shape geometry
- n `GXIntersectTransform`, which intersects the transform clip with a shape geometry
- n `GXDifferenceTransform`, which subtracts a shape geometry from the transform clip
- n `GXReverseDifferenceTransform`, which subtracts the transform clip from a shape geometry
- n The `GXExcludeTransform`, which performs an exclusive-OR operation with the transform clip and a shape geometry

GXUnionTransform

You can use the `GXUnionTransform` function to calculate the union of the geometry of the clip shape in a transform object with the geometry of a specified shape object, and then replace the transform's clip geometry with the resulting geometry.

```
void GXUnionTransform(gxTransform target, gxShape operand);
```

`target` A reference to the transform object containing the clip property you want to modify.

`operand` A reference to the shape containing the geometry you want to combine with the target transform's clip.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
transform_is_nil
shape_is_nil
number_of_contours_exceeds_implementation_limit
number_of_points_exceeds_implementation_limit
size_of_path_exceeds_implementation_limit
size_of_polygon_exceeds_implementation_limit
shapeFill_not_allowed (debugging version)
shape_access_not_allowed (debugging version)
clip_to_frame_shape_unimplemented (debugging version)
shape_may_not_be_a_bitmap (debugging version)
shape_may_not_be_a_picture (debugging version)
```

Warnings

```
character_substitution_took_place
unable_to_traverse_open_contour_that_starts_or_ends_off_the_curve (debugging version)
```

SEE ALSO

For an example of the use of this function, see page 6-23.

For an illustration of the effects of the `GXUnionTransform` function on the transform clip, see Figure 6-8 on page 6-22. For a general description of constructive geometry operations, see the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*.

GXIntersectTransform

You can use the `GXIntersectTransform` function to calculate the intersection of the geometry of the clip shape in a transform object with the geometry of a specified shape object, and then replace the transform's clip geometry with the resulting geometry.

```
void GXIntersectTransform(gxTransform target, gxShape operand);
```

`target` A reference to the transform object containing the clip property you want to modify.

`operand` A reference to the shape containing the geometry you want to intersect with the target transform's clip.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
transform_is_nil
shape_is_nil
number_of_contours_exceeds_implementation_limit
number_of_points_exceeds_implementation_limit
size_of_path_exceeds_implementation_limit
size_of_polygon_exceeds_implementation_limit
shapeFill_not_allowed (debugging version)
shape_access_not_allowed (debugging version)
clip_to_frame_shape_unimplemented (debugging version)
shape_may_not_be_a_bitmap (debugging version)
shape_may_not_be_a_picture (debugging version)
```

Warnings

```
character_substitution_took_place
unable_to_traverse_open_contour_that_starts_or_ends_off_the_curve (debugging version)
```

SEE ALSO

For an illustration of the effects of the `GXIntersectTransform` function on the transform clip, see Figure 6-8 on page 6-22. For a general description of constructive geometry operations, see the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*.

GXDifferenceTransform

You can use the `GXDifferenceTransform` function to subtract the geometry of a specified shape object from the geometry of the clip shape in a transform object, and then replace the transform's clip property with the resulting geometry.

```
void GXDifferenceTransform(gxTransform target, gxShape operand);
```

`target` A reference to the transform object containing the clip property you want to modify.

`operand` A reference to the shape containing the geometry you want to subtract from the target transform's clip.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
transform_is_nil
shape_is_nil
number_of_contours_exceeds_implementation_limit
number_of_points_exceeds_implementation_limit
size_of_path_exceeds_implementation_limit
size_of_polygon_exceeds_implementation_limit
shapeFill_not_allowed (debugging version)
shape_access_not_allowed (debugging version)
clip_to_frame_shape_unimplemented (debugging version)
shape_may_not_be_a_bitmap (debugging version)
shape_may_not_be_a_picture (debugging version)
```

Warnings

```
character_substitution_took_place
unable_to_traverse_open_contour_that_starts_or_ends_off_the_curve (debugging version)
```

SEE ALSO

For an illustration of the effects of the `GXDifferenceTransform` function on the transform clip, see Figure 6-8 on page 6-22. For a general description of constructive geometry operations, see the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*.

GXReverseDifferenceTransform

You can use the `GXReverseDifferenceTransform` function to subtract the geometry of the clip shape in a transform object from the geometry of a specified shape object, and then replace the transform's clip property with the resulting geometry.

```
void GXReverseDifferenceTransform(gxTransform target,
                                gxShape operand);
```

`target` A reference to the transform object containing the clip property you want to modify.

`operand` A reference to the shape containing the geometry from which you want to subtract the target transform's clip.

ERRORS, WARNINGS, AND NOTICES

Errors

<code>out_of_memory</code>	
<code>transform_is_nil</code>	
<code>shape_is_nil</code>	
<code>number_of_contours_exceeds_implementation_limit</code>	
<code>number_of_points_exceeds_implementation_limit</code>	
<code>size_of_path_exceeds_implementation_limit</code>	
<code>size_of_polygon_exceeds_implementation_limit</code>	
<code>shapeFill_not_allowed</code>	(debugging version)
<code>shape_access_not_allowed</code>	(debugging version)
<code>clip_to_frame_shape_unimplemented</code>	(debugging version)
<code>shape_may_not_be_a_bitmap</code>	(debugging version)
<code>shape_may_not_be_a_picture</code>	(debugging version)

Warnings

<code>character_substitution_took_place</code>	
<code>unable_to_traverse_open_contour_that_starts_or_ends_off_the_curve</code>	(debugging version)

SEE ALSO

For an illustration of the effects of the `GXReverseDifferenceTransform` function on the transform clip, see Figure 6-8 on page 6-22. For a general description of constructive geometry operations, see the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*.

GXExcludeTransform

You can use the `GXExcludeTransform` function to perform an exclusive-OR operation between the geometry of the clip shape in a transform object and the geometry of a specified shape object, and then replace the transform's clip property with the resulting geometry.

```
void GXExcludeTransform(gxTransform target, gxShape operand);
```

`target` A reference to the transform object containing the clip property you want to modify.

`operand` A reference to the shape containing the geometry you want to combine with the target transform's clip in an exclusive-OR operation.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
transform_is_nil
shape_is_nil
number_of_contours_exceeds_implementation_limit
number_of_points_exceeds_implementation_limit
size_of_path_exceeds_implementation_limit
size_of_polygon_exceeds_implementation_limit
shapeFill_not_allowed (debugging version)
shape_access_not_allowed (debugging version)
clip_to_frame_shape_unimplemented (debugging version)
shape_may_not_be_a_bitmap (debugging version)
shape_may_not_be_a_picture (debugging version)
```

Warnings

```
character_substitution_took_place
unable_to_traverse_open_contour_that_starts_or_ends_off_the_curve (debugging version)
```

SEE ALSO

For an illustration of the effects of the `GXExcludeTransform` function on the transform clip, see Figure 6-8 on page 6-22. For a general description of constructive geometry operations, see the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*.

Getting and Setting the Mapping

This section describes the functions you can use to manipulate a transform object's mapping matrix.

GXGetTransformMapping

You can use the `GXGetTransformMapping` function to retrieve the mapping property of a transform object.

```
gxMapping *GXGetTransformMapping(gxTransform source,
                                  gxMapping *map);
```

`source` A reference to the transform object whose mapping you want to examine.

`map` A pointer to a mapping structure. On return, the structure contains the mapping matrix of the source transform.

function result A pointer to the mapping property of the source transform. (This value is the same as the value returned in the `map` parameter.)

DESCRIPTION

The `GXGetTransformMapping` function copies the mapping matrix information from the mapping property of the source transform object into the mapping structure pointed to by the `map` parameter. The function also returns as its function result a pointer to this mapping structure.

Note that the returned mapping is a copy; you can alter it without affecting the mapping property of the source transform. If you call this function and alter the mapping that it returns, you can then assign that changed mapping back to the transform object by calling the `GXSetTransformMapping` function.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`transform_is_nil`
`parameter_is_nil` (debugging version)

SEE ALSO

For information about the mapping property of the transform object, see the section “Mapping” beginning on page 6-10. For information about mapping matrices in general, see the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

GXSetTransformMapping

You can use the `GXSetTransformMapping` function to assign a mapping to a transform object.

```
void GXSetTransformMapping(gxTransform target,
                          const gxMapping *map);
```

`target` A reference to the transform object you want to assign the mapping to.
`map` A pointer to a mapping structure containing the matrix you want to assign as the mapping property of the target transform.

DESCRIPTION

The `GXSetTransformMapping` function copies information from the mapping structure pointed to by the `map` parameter into the mapping property of the transform object referenced by the `target` parameter.

You can specify `nil` for the `map` parameter, in which case this function sets the mapping property of the target transform to the identity mapping. (An identity mapping has no transforming effect on shape geometries that it is applied to.)

SPECIAL CONSIDERATIONS

You may provide any values for the elements of the mapping structure pointed to by the `map` parameter, with one exception: the lower-right element of this matrix (element [2][2]) may not be 0.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`transform_is_nil`

Notices (debugging version)

`mapping_unaffected`

SEE ALSO

For information about the mapping property of the transform object, see the section “Mapping” beginning on page 6-10. For information about mapping matrices in general, see the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

GXGetShapeMapping

You can use the `GXGetShapeMapping` function to retrieve the mapping property of the transform object associated with a specified shape.

```
gxMapping *GXGetShapeMapping(gxShape source, gxMapping *map);
```

`source` A reference to the shape whose transform object contains the mapping property you want to examine.

`map` A pointer to the mapping structure. On return, the structure contains the mapping matrix of the source shape's transform.

function result A pointer to the mapping property of the source shape's transform. (This value is the same as the value returned in the `map` parameter.)

DESCRIPTION

The `GXGetShapeMapping` function copies the mapping matrix information from the mapping property of the source shape's transform object into the mapping structure pointed to by the `map` parameter. The function also returns as its function result a pointer to this mapping structure.

Note that the returned mapping is a copy; you can alter it without affecting the mapping property of the source shape's transform. If you call this function and alter the mapping that it returns, you can then assign that changed mapping back to the shape's transform object by calling the `GXSetShapeMapping` function.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`shape_is_nil`
`parameter_is_nil` (debugging version)

SEE ALSO

For information about the mapping property of the transform object, see the section "Mapping" beginning on page 6-10. For information about mapping matrices in general, see the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

GXSetShapeMapping

You can use the `GXSetShapeMapping` function to assign a new mapping to the transform object associated with a specified shape.

```
void GXSetShapeMapping(gxShape target, const gxMapping *map);
```

`target` A reference to the shape whose transform you want to assign the mapping to.

`map` A pointer to a mapping structure containing the matrix you want to assign as the mapping property of the target shape's transform.

DESCRIPTION

The `GXSetShapeMapping` function copies information from the mapping structure pointed to by the `map` parameter into the mapping property of the transform object associated with the shape referenced by the `target` parameter.

Calling this function is almost equivalent to

```
GXSetTransformMapping(GXGetShapeTransform(myShape), theMapping);
```

except that, if the source shape's transform object is shared with other shapes, `GXSetShapeMapping` creates a new copy of the transform object, attaches it to the source shape, and changes the mapping of the copy. That way, calling this function does not produce side effects on other shapes.

You can specify `nil` for the `map` parameter, in which case this function sets the mapping property of the target shape's transform to the identity matrix. (An identity mapping has no transforming effect on shape geometries that it is applied to.)

SPECIAL CONSIDERATIONS

You can provide any values for the elements of the mapping structure pointed to by the `map` parameter, with one exception: the lower-right element of this matrix (element `[2][2]`) may not be 0.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`shape_is_nil`

Notices (debugging version)

`mapping_unaffected`

SEE ALSO

To assign a mapping directly to a transform object, use the `GXSetTransformMapping` function, described on page 6-55.

For information about the mapping property of the transform object, see the section “Mapping” beginning on page 6-10. For information about mapping matrices in general, see the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

Transforming Shapes by Modifying Transform Mappings

The mapping property of transform objects allows you to perform sophisticated transformations to your shape’s geometries. To get and set the mapping property, use the functions described in the section “Getting and Setting the Mapping” beginning on page 6-53.

The functions described in this section perform the calculations necessary to achieve common matrix transformations, without you having to modify the mapping matrix directly:

- n `GXMoveTransform` alters a transform’s mapping to move a shape by a specified horizontal and vertical offset.
- n `GXMoveTransformTo` alters a transform’s mapping to move a shape to a specified position.
- n `GXScaleTransform` alters a transform’s mapping to scale a shape by specified horizontal and vertical factors around a specified origin.
- n `GXRotateTransform` alters a transform’s mapping to rotate a shape by a specified number of degrees around a specified origin.
- n `GXSkewTransform` alters a transform’s mapping to skew a shape by specified horizontal and vertical factors around a specified origin.
- n `GXMapTransform` concatenates (using matrix multiplication) a specified mapping matrix to the mapping matrix contained in a transform’s mapping property.

QuickDraw GX provides a corresponding set of functions that you can use to apply these common transformations directly to the geometry of a shape object, rather than to its transform mapping. They are described in the section “Transforming Shapes by Modifying Shape Geometries” beginning on page 6-65.

GXMoveTransform

You can use the `GXMoveTransform` function to alter the mapping property of a transform object so that it moves its associated shape by a specified horizontal and vertical distance.

```
void GXMoveTransform(gxTransform target, Fixed deltaX,
                    Fixed deltaY);
```

Transform Objects

<code>target</code>	A reference to the transform object whose mapping property you want to alter.
<code>deltaX</code>	The horizontal distance.
<code>deltaY</code>	The vertical distance.

DESCRIPTION

The `GXMoveTransform` function calculates a new mapping matrix for the transform object referenced by the `target` parameter. When applied to a shape, the new matrix performs the same mapping transformations on the shape as the original matrix, except that the new matrix also moves the shape horizontally by the distance specified in the `deltaX` parameter and vertically by the distance specified in the `deltaY` parameter.

The distances are specified in geometry space.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`transform_is_nil`

Warnings

`move_transform_out_of_range`

Notices (debugging version)

`mapping_unaffected`

SEE ALSO

For information about mapping matrices in general, see the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

To move a shape by altering its geometry, use the `GXMoveShape` function, described on page 6-66.

GXMoveTransformTo

You can use the `GXMoveTransformTo` function to alter the mapping property of a transform object so that it moves its associated shape to a specified position.

```
void GXMoveTransformTo(gxTransform target, Fixed x, Fixed y);
```

<code>target</code>	A reference to the transform object whose mapping property you want to alter.
<code>x</code>	The horizontal coordinate of the desired position.
<code>y</code>	The vertical coordinate of the desired position.

DESCRIPTION

The `GXMoveTransformTo` function calculates a new mapping matrix for the transform object referenced by the `target` parameter. When applied to a shape, the new mapping matrix performs the same mapping transformations on the shape as the original matrix, except that the new matrix moves the shape to the position specified by the `x` and `y` parameters.

The horizontal and vertical coordinates are specified in geometry space. However, the position they specify relates only to the translation values in the mapping itself; the values in a shape's geometry are added to these values to determine the shape's final position in local space.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`transform_is_nil`

Warnings

`move_transform_out_of_range`

Notices (debugging version)

`mapping_unaffected`

SEE ALSO

For an example of the use of this function, see page 6-24.

For information about mapping matrices in general, see the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

To move a shape to a specified position by altering its geometry, use the `GXMoveShapeTo` function, described on page 6-67.

GXScaleTransform

You can use the `GXScaleTransform` function to alter the mapping property of a transform object so that it scales its associated shape by specified horizontal and vertical factors about a specified origin.

```
void GXScaleTransform(gxTransform target, Fixed hScale,
                    Fixed vScale, Fixed xOffset, Fixed yOffset);
```

Transform Objects

<code>target</code>	A reference to the transform object whose mapping property you want to alter.
<code>hScale</code>	The horizontal scaling factor.
<code>vScale</code>	The vertical scaling factor.
<code>xOffset</code>	The horizontal coordinate of the origin to scale about.
<code>yOffset</code>	The vertical coordinate of the origin to scale about.

DESCRIPTION

The `GXScaleTransform` function calculates a new mapping matrix for the transform object referenced by the `target` parameter. When applied to a shape, the new mapping matrix performs the same mapping transformations on the shape as the original matrix, but the new matrix also scales the shape horizontally by the factor indicated by the `hScale` parameter and vertically by the factor indicated by the `vScale` parameter. The new matrix scales the shape about the origin specified by the `xOffset` and `yOffset` parameters. (The origin is the point whose coordinates do not change as a result of the scaling operation.)

A value of `ff(1)` for the `hScale` or `vScale` parameter indicates no change of scale in the corresponding direction.

The coordinates of the origin are specified in local space.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`transform_is_nil`

Warnings

`scale_transform_out_of_range`

Notices (debugging version)

`mapping_unaffected`

SEE ALSO

For examples of the use of this function, see page 6-17 and Listing 6-3 on page 6-25.

For information about mapping matrices in general, see the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

To scale a shape by altering its geometry, use the `GXScaleShape` function, described on page 6-68.

GXRotateTransform

You can use the `GXRotateTransform` function to alter the mapping property of a transform object so that it rotates its associated shape a specified number of degrees around a specified origin.

```
void GXRotateTransform(gxTransform target, Fixed degrees,
                      Fixed xOffset, Fixed yOffset);
```

<code>target</code>	A reference to the transform object whose mapping property you want to alter.
<code>degrees</code>	The amount to rotate.
<code>xOffset</code>	The horizontal coordinate of the origin to rotate around.
<code>yOffset</code>	The vertical coordinate of the origin to rotate around.

DESCRIPTION

The `GXRotateTransform` function calculates a new mapping matrix for the transform object referenced by the `target` parameter. When applied to a shape, the new mapping matrix performs the same mapping transformations on the shape as the original matrix, but the new matrix also rotates the shape by the number of degrees specified in the `degrees` parameter around the origin specified by the `xOffset` and `yOffset` parameters.

The coordinates of the origin are specified in local space.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`transform_is_nil`

Warnings

`rotate_transform_out_of_range`

Notices (debugging version)

`mapping_unaffected`

SEE ALSO

For an example of the use of this function, see Listing 6-3 on page 6-25.

For information about mapping matrices in general, see the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

To rotate a shape by altering its geometry, use the `GXRotateShape` function, described on page 6-70.

GXSkewTransform

You can use the `GXSkewTransform` function to alter the mapping property of a transform object so that it skews its associated shape about a specified origin by specified horizontal and vertical factors.

```
void GXSkewTransform(gxTransform target, Fixed xSkew,
                    Fixed ySkew, Fixed xOffset, Fixed yOffset);
```

<code>target</code>	A reference to the transform object whose mapping property you want to alter.
<code>hSkew</code>	The amount to skew in the horizontal direction.
<code>vSkew</code>	The amount to skew in the vertical direction.
<code>xOffset</code>	The horizontal coordinate of the origin to skew about.
<code>yOffset</code>	The vertical coordinate of the origin to skew about.

DESCRIPTION

The `GXSkewTransform` function calculates a new mapping matrix for the transform object referenced by the `target` parameter. When applied to a shape, the new mapping matrix performs the same mapping transformations on the shape as the original matrix, but the new matrix also skews the shape in the horizontal direction by the factor indicated by the `hSkew` parameter, and in the vertical direction by the factor indicated by the `vSkew` parameter. The new matrix skews the shape about the origin specified by the `xOffset` and `yOffset` parameters. (The origin is the point whose coordinates do not change as a result of the scaling operation.)

The skew factors are expressed as a proportional amount of shift in one direction with distance in the perpendicular direction. A value of 0 for the `hSkew` or `vSkew` parameter indicates no skewing in the corresponding direction.

The coordinates of the origin are specified in local space.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
transform_is_nil
```

Warnings

```
skew_transform_out_of_range
```

Notices (debugging version)

```
mapping_unaffected
```

SEE ALSO

For an example of the use of this function, see Listing 6-3 on page 6-25.

For information about mapping matrices in general, see the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

To skew a shape by altering its geometry, use the `GXSkewShape` function, described on page 6-71.

GXMapTransform

You can use the `GXMapTransform` function to apply a separate mapping matrix to the mapping of a transform object, so that its associated shape is additionally transformed according to the specifications of the matrix.

```
void GXMapTransform(gxTransform target, const gxMapping *map);
```

`target` A reference to the transform object whose mapping property you want to alter.

`map` A pointer to a mapping structure containing the information you want to incorporate into the target transform's mapping.

DESCRIPTION

The `GXMapTransform` function calculates a new mapping matrix for the transform object referenced by the `target` parameter. It does so by concatenating the mappings (performing matrix multiplication). When applied to a shape, the transform's new mapping matrix performs the same transformations as the transform's original matrix as well as the transformations indicated by the matrix pointed to by the `map` parameter.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`transform_is_nil`

Warnings

`map_transform_out_of_range`

Notices (debugging version)

`mapping_unaffected`

SEE ALSO

For information about mapping matrices in general, see the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

To use a mapping matrix to alter a shape's geometry, use the `GXMapShape` function, described on page 6-72.

Transforming Shapes by Modifying Shape Geometries

The functions described in this section perform the calculations necessary to achieve common matrix transformations of shapes. They are equivalent to the functions that modify transform mappings as described in the previous section, "Transforming Shapes by Modifying Transform Mappings" beginning on page 6-58; however, the functions in this section can perform their transformations by directly modifying a shape's geometry rather than altering its transform's mapping:

- n `GXMoveShape` alters a shape's geometry to move it by a specified horizontal and vertical offset.
- n `GXMoveShapeTo` alters a shape's geometry to move it to a specified position.
- n `GXScaleShape` alters a shape's geometry to scale it by specified horizontal and vertical factors around a specified origin.
- n `GXRotateShape` alters a shape's geometry to rotate it by a specified number of degrees around a specified origin.
- n `GXSkewShape` alters a shape's geometry to skew it by specified horizontal and vertical factors around a specified origin.
- n `GXMapShape` alters a shape's geometry by applying a specified mapping matrix to it.

(Note also that the function `GXSetShapeBounds` works in the same manner as the other functions listed here; it modifies a shape's geometry to effect a scaling operation.

However, `GXSetShapeBounds` is described in the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*.)

When applied to a shape object, each of these functions performs its transformation in one of two ways:

- n If the shape's `gxMapTransformShape` attribute is cleared, these functions apply their mapping operations directly to the points of the shape object's geometry. In this case, the shape's transform mapping is unaffected.
- n If the shape's `gxMapTransformShape` attribute is set, these functions apply their mapping operations by altering the mapping property of the shape's transform object, in the manner of the transform-altering functions described in the previous section. In this case, the shape's geometry is unaffected.

GXMoveShape

You can use the `GXMoveShape` function to move a shape by a specified horizontal and vertical distance.

```
void GXMoveShape(gxShape target, Fixed deltaX, Fixed deltaY);
```

`target` A reference to the shape you want to move.

`deltaX` The horizontal distance to move the shape.

`deltaY` The vertical distance to move the shape.

DESCRIPTION

The `GXMoveShape` function changes the position of the shape referenced by the `target` parameter horizontally by the distance specified in the `deltaX` parameter and vertically by the distance specified in the `deltaY` parameter.

This function moves the target shape by the specified offsets in one of two ways:

- n If the target shape's `gxMapTransformShape` attribute is cleared, the function recalculates the control points of the shape's geometry to effect the move.
- n If the target shape's `gxMapTransformShape` attribute is set, this function is identical to the `GXMoveTransform` function; it recalculates the mapping matrix of the target shape's transform object to effect the move. If the target shape shares this transform object with other shapes, QuickDraw GX makes a copy of the transform object, associates the copy with the target shape, and makes changes to the copy.

The target shape can be any shape type. However, if the target shape is an empty shape, a full shape, or a picture shape, this function has no effect unless the shape's `gxMapTransformShape` attribute is set. Also, if the shape is a text, glyph, or layout shape and it contains no characters, this function has no effect.

The distances are specified in geometry coordinates.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`shape_is_nil`
`shape_access_not_allowed` (debugging version)

Warnings

`move_shape_out_of_range`
`graphic_type_cannot_be_moved`

Notices (debugging version)

`mapping_unaffected`

SEE ALSO

To move a shape by altering the mapping property of its transform object, you can also use the `GXMoveTransform` function, described on page 6-58.

GXMoveShapeTo

You can use the `GXMoveShapeTo` function to move a shape to a specified position.

```
void GXMoveShapeTo(gxShape target, Fixed x, Fixed y);
```

<code>target</code>	A reference to the shape you want to move.
<code>x</code>	The horizontal coordinate of the position to move the shape to.
<code>y</code>	The vertical coordinate of the position to move the shape to.

DESCRIPTION

The `GXMoveShapeTo` function moves the shape referenced by the `target` parameter to the position specified by the `x` and `y` parameters. The position corresponds to a specific point in the shape's geometry:

- n For point, line, and curve shapes, the point (`x`, `y`) corresponds to the first point in the shape's geometry.
- n For rectangle, polygon, path, and bitmap shapes, the point (`x`, `y`) corresponds to the top-left corner of the bounding rectangle.
- n For text, glyph, and layout shapes, the point (`x`, `y`) corresponds to the origin of the first glyph. If the shape contains no characters, this function has no effect.
- n Other shapes (empty shapes, full shapes, and pictures) cannot be moved.

This function relocates the target shape in one of two ways:

- n If the target shape's `gxMapTransformShape` attribute is cleared, the function recalculates the control points of the shape's geometry to effect the move.
- n If the target shape's `gxMapTransformShape` attribute is set, this function is identical to the `GXMoveTransformTo` function; it recalculates the mapping matrix of the target shape's transform object to effect the move. If the target shape shares this transform object with other shapes, QuickDraw GX makes a copy of the transform object, associates the copy with the target shape, and makes changes to the copy.

The target shape can be any shape type. However, if the target shape is an empty shape, a full shape, or a picture shape, this function has no effect unless the shape's `gxMapTransformShape` attribute is set.

The horizontal and vertical coordinates are specified in geometry space.

SPECIAL CONSIDERATIONS

This function does not necessarily move the target shape to the position in local space specified by the `x` and `y` parameters. Furthermore, if the shape's `gxMapTransformShape` attribute is set, this function does not necessarily move the shape to the same position it would if the `gxMapTransformShape` attribute were cleared:

- n With the attribute cleared, this function modifies shape geometry so that, in geometry space, the shape is at the position specified in the `x` and `y` parameters. However, the function ignores the transform mapping, so the shape's resultant position in local space will be at (x, y) only if the transform mapping specifies no translation.
- n With the attribute set, this function ignores shape geometry and sets the translation values in the shape's transform mapping to reflect the `x` and `y` parameters. Thus the shape's resultant position in local space will be at (x, y) only if its position in geometry space is at $(0.0, 0.0)$.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`shape_is_nil`
`shape_access_not_allowed` (debugging version)

Warnings

`move_shape_out_of_range`
`graphic_type_cannot_be_moved`

Notices (debugging version)

`mapping_unaffected`

SEE ALSO

For an example of the use of this function, see page 6-27.

To move a shape to a specified position by altering the mapping property of its transform object, you can also use the `GXMoveTransformTo` function, described on page 6-59.

GXScaleShape

You can use the `GXScaleShape` function to scale a shape by specified horizontal and vertical factors about a specified origin.

```
void GXScaleShape(gxShape target, Fixed hScale, Fixed vScale,
                 Fixed xOffset, Fixed yOffset);
```

`target` A reference to the shape you want to scale.
`hScale` The horizontal scaling factor.

Transform Objects

<code>vScale</code>	The vertical scaling factor.
<code>xOffset</code>	The horizontal coordinate of the origin to scale the shape about.
<code>yOffset</code>	The vertical coordinate of the origin to scale the shape about.

DESCRIPTION

The `GXScaleShape` function scales the shape referenced by the `target` parameter horizontally by the factor specified in the `hScale` parameter and vertically by the factor specified in the `vScale` parameter. The scaling is centered about the origin specified in the `xOffset` and `yOffset` parameters. (The origin is the point whose coordinates do not change as a result of the scaling operation.)

This function scales the target shape in one of two ways:

- n If the target shape's `gxMapTransformShape` attribute is cleared, the function recalculates the control points of the shape's geometry to effect the scaling.
- n If the target shape's `gxMapTransformShape` attribute is set, this function is identical to the `GXScaleTransform` function; it recalculates the mapping matrix of the target shape's transform object to effect the scaling. If the target shape shares this transform object with other shapes, QuickDraw GX makes a copy of the transform object, associates the copy with the target shape, and makes changes to the copy.

The target shape can be any shape type. However, if the target shape is an empty shape, a full shape, or a picture shape, this function has no effect unless the shape's `gxMapTransformShape` attribute is set.

The coordinates of the origin are specified in geometry space.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`shape_is_nil`
`shape_access_not_allowed` (debugging version)

Warnings

`scale_shape_out_of_range`
`graphic_type_cannot_be_moved`

Notices (debugging version)

`mapping_unaffected`

SEE ALSO

For examples of the use of this function, see Listing 6-4 on page 6-25 and Listing 6-5 on page 6-27.

To scale a shape by altering the mapping property of its transform object, you can also use the `GXScaleTransform` function, described on page 6-60.

GXRotateShape

You can use the `GXRotateShape` function to rotate a shape around a specified origin.

```
void GXRotateShape(gxShape target, Fixed degrees,
                  Fixed xOffset, Fixed yOffset);
```

`target` A reference to the shape you want to rotate.
`degrees` The number of degrees to rotate the shape.
`xOffset` The horizontal coordinate of the origin to rotate the shape around.
`yOffset` The vertical coordinate of the origin to rotate the shape around.

DESCRIPTION

The `GXRotateShape` function rotates the shape referenced by the `target` parameter by the number of degrees specified in the `degrees` parameter around the origin specified by the `xOffset` and `yOffset` parameters.

This function rotates the target shape in one of two ways:

- n If the target shape's `gxMapTransformShape` attribute is cleared, the function recalculates the control points of the shape's geometry to effect the rotation.
- n If the target shape's `gxMapTransformShape` attribute is set, this function is identical to the `GXRotateTransform` function; it recalculates the mapping matrix of the target shape's transform object to effect the rotation. If the target shape shares this transform object with other shapes, QuickDraw GX makes a copy of the transform object, associates the copy with the target shape, and makes changes to the copy.

The target shape can be any shape type. However, if the target shape is an empty shape, a full shape, or a picture shape, this function has no effect unless the shape's `gxMapTransformShape` attribute is set.

The coordinates of the origin are specified in geometry space.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`shape_is_nil`
`shape_access_not_allowed` (debugging version)

Warnings

`rotate_shape_out_of_range`
`graphic_type_cannot_be_moved`

Notices (debugging version)

`mapping_unaffected`

SEE ALSO

For an example of the use of this function, see Listing 6-4 on page 6-25.

To rotate a shape by altering the mapping property of its transform object, you can also use the `GXRotateTransform` function, described on page 6-62.

GXSkewShape

You can use the `GXSkewShape` function to skew a shape about a specified origin by specified horizontal and vertical factors.

```
void GXSkewShape(gxShape target, Fixed xSkew, Fixed ySkew,
                 Fixed xOffset, Fixed yOffset);
```

<code>target</code>	A reference to the shape you want to skew.
<code>hSkew</code>	The amount to skew the shape horizontally.
<code>vSkew</code>	The amount to skew the shape vertically.
<code>xOffset</code>	The horizontal coordinate of the origin to skew the shape about.
<code>yOffset</code>	The vertical coordinate of the origin to skew the shape about.

DESCRIPTION

The `GXSkewShape` function skews the shape referenced by the `target` parameter horizontally by the factor specified in the `hSkew` parameter and vertically by the factor specified in the `vSkew` parameter. The skewing is centered about the origin specified in the `xOffset` and `yOffset` parameters. (The origin is the point whose coordinates do not change as a result of the skewing operation.)

The skew factors are expressed as a proportional amount of shift in one direction with distance in the perpendicular direction. A value of 0 for the `hSkew` or `vSkew` parameter indicates no skewing in the corresponding direction.

This function skews the target shape in one of two ways:

- n If the target shape's `gxMapTransformShape` attribute is cleared, the function recalculates the control points of the shape's geometry to effect the skewing.
- n If the target shape's `gxMapTransformShape` attribute is set, this function is identical to the `GXSkewTransform` function; it recalculates the mapping matrix of the target shape's transform object to effect the skewing. If the target shape shares this transform object with other shapes, QuickDraw GX makes a copy of the transform object, associates the copy with the target shape, and makes changes to the copy.

The target shape can be any shape type. However, if the target shape is an empty shape, a full shape, or a picture shape, this function has no effect unless the shape's `gxMapTransformShape` attribute is set.

The coordinates of the origin are specified in geometry space.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`shape_is_nil`
`shape_access_not_allowed` (debugging version)

Warnings

`skew_shape_out_of_range`
`graphic_type_cannot_be_moved`

Notices (debugging version)

`mapping_unaffected`

SEE ALSO

For examples of the use of this function, see Listing 6-4 on page 6-25 and Listing 6-5 on page 6-27.

To skew a shape by altering the mapping property of its transform object, you can also use the `GXSkewTransform` function, described on page 6-63.

GXMapShape

You can use the `GXMapShape` function to apply an arbitrary mapping matrix to a shape.

```
void GXMapShape(gxShape target, const gxMapping *map);
```

target A reference to the shape you want to apply the mapping to.
map A pointer to a mapping structure containing the matrix you want to apply to the target shape.

DESCRIPTION

The `GXMapShape` function applies to the target shape the mapping transformations represented by the mapping matrix pointed to by the `map` parameter.

This function applies the mapping in one of two ways:

- n If the target shape's `gxMapTransformShape` attribute is cleared, the function applies the mapping matrix directly to the points of the shape's geometry.
- n If the target shape's `gxMapTransformShape` attribute is set, this function is identical to the `GXMapTransform` function; it concatenates the target shape's transform mapping with the mapping matrix pointed to by the `map` parameter. If the target shape shares its transform object with other shapes, QuickDraw GX makes a copy of the transform object, associates the copy with the target shape, and makes changes to the copy.

The target shape can be any shape type. However, if the target shape is an empty shape, a full shape, or a picture shape, this function has no effect unless the shape's `gxMapTransformShape` attribute is set.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`shape_is_nil`
`shape_access_not_allowed` (debugging version)

Warnings

`map_shape_out_of_range`
`graphic_type_cannot_be_moved`

Notices (debugging version)

`mapping_unaffected`

SEE ALSO

To apply a mapping matrix to the mapping property of a transform object, you can also use the `GXMapTransform` function, described on page 6-64.

Getting and Setting the View Port List

This section describes the functions you can use to manipulate the view port list property of a transform object. For information about the view port list property, see “View Port List” on page 6-11. For general information about view ports, see the chapter “View-Related Objects” in this book.

GXGetTransformViewPorts

You can use the `GXGetTransformViewPorts` function to retrieve the view port list of a transform object.

```
long GXGetTransformViewPorts(gxTransform source,
                             gxViewPort list[]);
```

source A reference to the transform object whose view port list you want to examine.

list An array of view port references. On return, the array contains the view port list of the source transform.

function result The number of view ports in the source transform's view port list.

DESCRIPTION

The `GXGetTransformViewPorts` function copies the list of view port references in the source transform into the array referenced by the `list` parameter, and returns as the function result the total number of view port references in the list.

If you pass `nil` for the `list` parameter, the function returns the number of view ports as the function result, but does not return the references to the view ports. Thus you normally call this function twice: once to determine the size of view port array to allocate, and once to retrieve the array itself.

SPECIAL CONSIDERATIONS

This function returns all view port references in the source transform's view port list, including any invalid ones (for view ports that have been disposed of).

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`transform_is_nil`

Notices (debugging version)

`transform_references_disposed_viewPort`

SEE ALSO

For an example of the use of this function, see Listing 6-6 on page 6-29.

To assign a view port list to a transform object, use the `GXSetTransformViewPorts` function, described next.

To retrieve the view port list of the transform object associated with a specified shape, use the `GXGetShapeViewPorts` function, described on page 6-75.

GXSetTransformViewPorts

You can use the `GXSetTransformViewPorts` function to assign a view port list to a transform object.

```
void GXSetTransformViewPorts(gxTransform target, long count,
                             const gxViewPort list[]);
```

<code>target</code>	A reference to the transform object to which you want to assign the view port list.
<code>count</code>	The number of entries in the new view port list; the size of the <code>list</code> array.
<code>list</code>	The new view port list; an array of references to the view ports you want to associate with the source transform.

DESCRIPTION

The `GXSetTransformViewPorts` function replaces the view port list of the transform object referenced by the `target` parameter with the view port list contained in the `list` parameter. The `count` parameter specifies the number of view ports in the new list.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`transform_is_nil`
`invalid_viewPort_reference`
`parameter_out_of_range` (debugging version)

Notices (debugging version)

`transform_viewPorts_already_set`

SEE ALSO

For an example of the use of this function, see Listing 6-6 on page 6-29.

To retrieve the view port list of a transform object, use the `GXGetTransformViewPorts` function, described in the previous section.

To assign a view port list to the transform object associated with a specified shape, use the `GXSetShapeViewPorts` function, described on page 6-76.

GXGetShapeViewPorts

You can use the `GXGetShapeViewPorts` function to retrieve the view port list of the transform object associated with a specific shape.

```
long GXGetShapeViewPorts(gxShape source,
                        gxViewPort list[]);
```

`source` A reference to the shape whose transform object contains the view port list you want to examine.

`list` An array of view port references. On return, the array contains the view port list of the source shape's transform.

function result The number of references in the view port list of the source shape's transform.

DESCRIPTION

The `GXGetShapeViewPorts` function copies references to the view ports associated with the source shape's transform into the array referenced by the `list` parameter, and returns as the function result the total number of view port references in the list.

If you pass `nil` for the `list` parameter, the function returns the number of view ports as the function result, but does not return the references to the view ports. Thus you normally call this function twice: once to determine the size of view port array to allocate, and once to retrieve the array itself.

SPECIAL CONSIDERATIONS

This function returns all view port references in the source transform's view port list, including any invalid ones (for view ports that have been disposed of).

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`shape_is_nil`

Notices (debugging version)

`transform_references_disposed_viewPort`

SEE ALSO

To assign a view port list to the transform object associated with a specified shape, use the `GXSetShapeViewPorts` function, described next.

To retrieve the view port list of a transform object, use the `GXGetTransformViewPorts` function, described on page 6-73.

GXSetShapeViewPorts

You can use the `GXSetShapeViewPorts` function to assign a view port list to the transform object associated with a particular shape.

```
void GXSetShapeViewPorts(gxShape target, long count,
                        const gxViewPort list[]);
```

<code>target</code>	A reference to the shape object whose transform's view port list you want to replace.
<code>count</code>	The number of view port references in the new view port list; the size of the <code>list</code> array.
<code>list</code>	The new view port list; an array of references to the view ports you want to associate with the source shape's transform.

DESCRIPTION

The `GXSetShapeViewPorts` function replaces the view port list of the transform object associated with the shape referenced by the `target` parameter with the view port list specified by the `list` parameter. The `count` parameter specifies the number of view ports in the new list.

If the source shape shares its transform object with other shapes, this function first copies the transform, associates the copy with the source shape, and then makes changes to the copy.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`shape_is_nil`
`invalid_viewPort_reference`
`parameter_out_of_range` (debugging version)

Notices (debugging version)

`transform_viewPorts_already_set`

SEE ALSO

To retrieve the view port list from the transform object associated with a specified shape, use the `GXGetShapeViewPorts` function, described in the previous section.

To assign a view port list directly to a transform object, use the `GXSetTransformViewPorts` function, described on page 6-74.

Getting and Setting the Hit-Test Parameters

This section describes the functions you can use to manipulate the hit-test parameters property of a transform object. For general information on the hit-test parameters property, see the section “Hit-Test Parameters” beginning on page 6-11.

For information on hit-testing and using the `GXHitTestShape` or `GXHitTestPicture` functions, see the chapter “Shape Objects” in this book. For additional information on `GXHitTestPicture`, see the picture shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*.

GXGetTransformHitTest

You can use the `GXGetTransformHitTest` function to retrieve the hit-test parameters of a transform object.

```
gxShapePart GXGetTransformHitTest(gxTransform source,
                                   Fixed *tolerance);
```

`source` A reference to the transform object whose hit-test parameters you want to examine.

`tolerance` A pointer to a `Fixed` value. On return, the value specifies the hit-test tolerance of the source transform.

function result The shape-parts mask of the source transform, specifying the shape parts to be tested for.

DESCRIPTION

The `GXGetTransformHitTest` function returns the contents of the hit-test parameters property of the transform object referenced by the `source` parameter. The shape-parts mask is returned as the function result and the hit-test tolerance is returned by the `tolerance` parameter.

Hit-test tolerance is specified in geometry units. You can specify `nil` for the `tolerance` parameter, in which case the hit-test tolerance is not returned.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`transform_is_nil`

SEE ALSO

For information about the hit-test parameters property, see “Hit-Test Parameters” beginning on page 6-11. To interpret the values in the shape-parts mask, see Table 6-1 on page 6-12.

To assign hit-test parameters to a transform object, use the `GXSetTransformHitTest` function, described next.

To retrieve the hit-test parameters of the transform associated with a specified shape, use the `GXGetShapeHitTest` function, described on page 6-80.

GXSetTransformHitTest

You can use the `GXSetTransformHitTest` function to assign new hit-test parameters to a transform object.

```
void GXSetTransformHitTest(gxTransform target, gxShapePart mask,
                          Fixed tolerance);
```

<code>target</code>	A reference to the transform object whose hit-test parameters you want to assign.
<code>mask</code>	The shape-parts mask to assign to the target transform.
<code>tolerance</code>	The hit-test tolerance to assign to the target transform. It is measured in geometry units, and can be 0 or any positive number.

DESCRIPTION

The `GXSetTransformHitTest` function assigns the shape-parts mask contained in the `mask` parameter and the hit-test tolerance contained in the `tolerance` parameter to the transform object referenced by the `target` parameter. The tolerance value cannot be negative.

ERRORS, WARNINGS, AND NOTICES

Errors

<code>out_of_memory</code>	
<code>transform_is_nil</code>	
<code>parameter_out_of_range</code>	(debugging version)
<code>tolerance_out_of_range</code>	(debugging version)

SEE ALSO

For information about the hit-test parameters property, see “Hit-Test Parameters” beginning on page 6-11. To interpret the values in the shape-parts mask, see Table 6-1 on page 6-12.

To retrieve the hit-test parameters of a transform object, use the `GXGetTransformHitTest` function, described in the previous section.

To assign hit-test parameters to the transform associated with a specified shape, use the `GXSetShapeHitTest` function, described on page 6-81.

GXGetShapeHitTest

You can use the `GXGetShapeHitTest` function to retrieve the hit-test parameters of the transform object associated with a particular shape.

```
gxShapePart GXGetShapeHitTest(gxShape source,
                               Fixed *tolerance);
```

`source` A reference to the shape whose transform object contains the hit-test parameters you want to examine.

`tolerance` A pointer to a `Fixed` value. On return, the value specifies the hit-test tolerance of the source shape's transform.

function result The shape-parts mask of the source shape's transform.

DESCRIPTION

The `GXGetShapeHitTest` function returns the hit-test parameters from the transform object associated with the shape referenced by the `source` parameter. The shape-parts mask is returned as the function result and the hit-test tolerance is returned by the `tolerance` parameter.

Hit-test tolerance is specified in geometry units. You can specify `nil` for the `tolerance` parameter, in which case the hit-test tolerance is not returned.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`shape_is_nil`

SEE ALSO

For information about the hit-test parameters property, see “Hit-Test Parameters” beginning on page 6-11. To interpret the values in the shape-parts mask, see Table 6-1 on page 6-12.

To assign hit-test parameters to the transform associated with a specified shape, use the `GXSetShapeHitTest` function, described next.

To retrieve the hit-test parameters of a specified transform object, use the `GXGetTransformHitTest` function, described on page 6-78.

GXSetShapeHitTest

You can use the `GXSetShapeHitTest` function to assign hit-test parameters to the transform object associated with a particular shape.

```
void GXSetShapeHitTest(gxShape target, gxShapePart mask,
                       Fixed tolerance);
```

`target` A reference to the shape associated with the transform object whose hit-test parameters you want to assign.

`mask` The shape-parts mask to assign to the transform.

`tolerance` The hit-test tolerance to assign to the transform. It is measured in geometry units, and can be 0 or any positive number.

DESCRIPTION

The `GXSetShapeHitTest` function assigns the shape-parts mask contained in the `mask` parameter and the hit-test tolerance contained in the `tolerance` parameter to the transform object associated with the shape referenced by the `target` parameter. The tolerance value cannot be negative.

If the `target` shape shares its transform object with other shapes, this function first makes a copy of the transform object, associates it with the `target` shape, and then assigns the new hit-test parameters to the copy.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
shape_is_nil
parameter_out_of_range      (debugging version)
tolerance_out_of_range      (debugging version)
```

SEE ALSO

For an example of the use of this function, see page 6-30.

For information about the hit-test parameters property, see “Hit-Test Parameters” beginning on page 6-11. To interpret the values in the shape-parts mask, see Table 6-1 on page 6-12.

To retrieve the hit-test parameters from the transform associated with a specified shape, use the `GXGetShapeHitTest` function, described in the previous section.

To assign hit-test parameters directly to a transform object, use the `GXSetTransformHitTest` function, described on page 6-79.

Summary of Transform Objects

Constants and Data Types

The Transform Object

```
typedef struct gxPrivateTransformRecord *gxTransform;
```

Shape Parts for Hit-Testing

```
enum gxShapeParts {          /* (in order of evaluation) */
    gxNoPart                 = 0,
    gxBoundsPart             = 0x0001,
    gxGeometryPart           = 0x0002,
    gxPenPart                 = 0x0004,
    gxCornerPointPart        = 0x0008,
    gxControlPointPart       = 0x0010,
    gxEdgePart                = 0x0020,
    gxJoinPart                = 0x0040,
    gxStartCapPart           = 0x0080,
    gxEndCapPart              = 0x0100,
    gxDashPart                = 0x0200,
    gxPatternPart            = 0x0400,
    gxGlyphBoundsPart        = gxJoinPart,
    gxGlyphFirstPart         = gxStartCapPart,
    gxGlyphLastPart          = gxEndCapPart,
    gxSideBearingPart        = gxDashPart,
    gxAnyPart                 = gxBoundsPart | gxGeometryPart |
        gxPenPart | gxCornerPointPart | gxControlPointPart |
        gxEdgePart | gxJoinPart | gxStartCapPart |
        gxEndCapPart | gxDashPart | gxPatternPart
};

typedef long gxShapePart;
```

Functions

Creating and Manipulating Transform Objects

```

gxTransform GXNewTransform    (void);
void GXDisposeTransform      (gxTransform target);
void GXCopyToTransform       (gxTransform target, gxTransform source);
boolean GXEqualTransform     (gxTransform one, gxTransform two);
gxTransform GXCloneTransform(gxTransform source);

```

Manipulating Transform Object Properties

```

void GXResetTransform        (gxTransform target);
long GXGetTransformOwners    (gxTransform source);
long GXGetTransformTags     (gxTransform source, long tagType,
                             long index, long count, gxTag items[]);
void GXSetTransformTags     (gxTransform target, long tagType,
                             long index, long oldCount, long newCount,
                             const gxTag items[]);

```

Getting and Setting a Transform's Clip

```

void GXSetTransformClip     (gxTransform target, gxShape clip);
gxShape GXGetTransformClip  (gxTransform source);
gxShape GXGetShapeClip      (gxShape source);
void GXSetShapeClip         (gxShape target, gxShape clip);

```

Performing Geometric Arithmetic on Transform Clips

```

void GXUnionTransform       (gxTransform target, gxShape operand);
void GXIntersectTransform   (gxTransform target, gxShape operand);
void GXDifferenceTransform  (gxTransform target, gxShape operand);
void GXReverseDifferenceTransform
                             (gxTransform target, gxShape operand);
void GXExcludeTransform     (gxTransform target, gxShape operand);

```

Getting and Setting a Transform's Mapping

```

gxMapping *GXGetTransformMapping
            (gxTransform source, gxMapping *map);
void GXSetTransformMapping (gxTransform target const gxMapping *map);
gxMapping *GXGetShapeMapping(gxShape source, gxMapping *map);
void GXSetShapeMapping     (gxShape target, const gxMapping *map);

```

Transforming Shapes by Modifying Transform Mappings

```

void GXMoveTransform      (gxTransform target, Fixed deltaX,
                          Fixed deltaY);

void GXMoveTransformTo   (gxTransform target, Fixed x, Fixed y);

void GXScaleTransform    (gxTransform target, Fixed hScale,
                          Fixed vScale, Fixed xOffset, Fixed yOffset);

void GXRotateTransform   (gxTransform target, Fixed degrees,
                          Fixed xOffset, Fixed yOffset);

void GXSkewTransform     (gxTransform target, Fixed xSkew,
                          Fixed ySkew, Fixed xOffset, Fixed yOffset);

void GXMapTransform      (gxTransform target, const gxMapping *map);

```

Transforming Shapes by Modifying Shape Geometries

```

void GXMoveShape         (gxShape target, Fixed deltaX, Fixed deltaY);

void GXMoveShapeTo      (gxShape target, Fixed x, Fixed y);

void GXScaleShape       (gxShape target, Fixed hScale, Fixed vScale,
                          Fixed xOffset, Fixed yOffset);

void GXRotateShape      (gxShape target, Fixed degrees,
                          Fixed xOffset, Fixed yOffset);

void GXSkewShape        (gxShape target, Fixed xSkew, Fixed ySkew,
                          Fixed xOffset, Fixed yOffset);

void GXMapShape         (gxShape target, const gxMapping *map);

```

Getting and Setting a Transform's View Ports

```

long GXGetTransformViewPorts(gxTransform source, gxViewPort list[]);

void GXSetTransformViewPorts(gxTransform target, long count,
                              const gxViewPort list[]);

long GXGetShapeViewPorts   (gxShape source, gxViewPort list[]);

void GXSetShapeViewPorts  (gxShape target, long count,
                              const gxViewPort list[]);

```

Getting and Setting a Transform's Hit-Test Parameters

```

gxShapePart GXGetTransformHitTest
              (gxTransform source, Fixed *tolerance);

void GXSetTransformHitTest (gxTransform target, gxShapePart mask,
                          Fixed tolerance);

gxShapePart GXGetShapeHitTest
              (gxShape source, Fixed *tolerance);

void GXSetShapeHitTest    (gxShape target, gxShapePart mask,
                          Fixed tolerance);

```

View-Related Objects

Contents

About View Ports, View Devices, and View Groups	7-5
About View Port Objects	7-7
View Port Properties	7-7
View Port Clip and Mapping	7-9
Dither	7-10
Halftone	7-13
Parent and Child View Ports	7-18
View Port Attributes	7-20
The Default View Port Object	7-20
View Port Objects and Windows	7-21
About View Device Objects	7-24
View Device Properties	7-25
View Device Clip and Mapping	7-26
View Device Bitmap	7-26
View Device Attributes	7-27
The Default View Device Object	7-28
View Device Objects and Physical Devices	7-28
About View Group Objects	7-29
View Groups Have No Properties	7-29
Onscreen and Offscreen View Groups	7-29
About Drawing, Coordinate Conversion, and Clipping	7-30
QuickDraw GX Coordinates	7-31
Geometry Space	7-32
Local Space	7-33
Global Space	7-34
Device Space	7-38

Using View-Related Objects	7-39
Using View Ports	7-40
Creating and Manipulating View Port Objects	7-40
Manipulating View Port Object Properties	7-42
Getting and Setting a View Port's Clip and Mapping	7-44
Setting Up the View Port Hierarchy for a Window	7-46
Supporting Scrolling in a Window	7-47
Identifying a View Port's View Devices	7-49
Identifying a Shape's View Ports	7-50
Measuring a Shape in Local Space	7-51
Using View Devices	7-52
Creating and Manipulating View Device Objects	7-52
Manipulating View Device Object Properties	7-54
Getting and Setting a View Device's Clip and Mapping	7-56
Identifying a Shape's View Devices	7-58
Measuring a Shape in Device Space	7-59
Hit-Testing a Shape on a Device	7-60
Using View Groups	7-60
Creating and Manipulating View Group Objects	7-61
Setting Up an Offscreen View Group	7-62
Measuring a Shape in Global Space	7-63
View-Related Objects Reference	7-65
Constants and Data Types	7-65
The View Port Object	7-65
The Halftone Structure	7-65
Dot Types	7-66
Tint Types	7-67
View Port Attributes	7-68
The View Device Object	7-68
View Device Attributes	7-68
The View Group Object	7-69
View Group Types	7-69
View Port Functions	7-69
Creating and Manipulating View Port Objects	7-70
GXNewViewPort	7-70
GXDisposeViewPort	7-71
GXCopyToViewPort	7-72
GXEqualViewPort	7-73
Manipulating View Port Object Properties	7-74
GXGetViewPortClip	7-74
GXSetViewPortClip	7-75
GXGetViewPortMapping	7-77
GXSetViewPortMapping	7-78
GXGetViewPortGlobalMapping	7-79
GXGetViewPortDither	7-80
GXSetViewPortDither	7-80
GXGetViewPortHalftone	7-81

GXSetViewPortHalftone	7-82
GXGetHalftoneDeviceAngle	7-83
GXGetViewPortParent	7-84
GXSetViewPortParent	7-84
GXGetViewPortChildren	7-86
GXSetViewPortChildren	7-87
GXGetViewPortViewGroup	7-88
GXSetViewPortViewGroup	7-88
GXGetViewPortAttributes	7-89
GXSetViewPortAttributes	7-90
GXGetViewPortTags	7-91
GXSetViewPortTags	7-92
Retrieving the View Devices That Intersect a View Port	7-94
GXGetViewPortViewDevices	7-94
Retrieving the View Ports That Intersect a Shape	7-95
GXGetShapeGlobalViewPorts	7-95
Measuring a Shape in Local Coordinates	7-96
GXGetShapeLocalBounds	7-96
View Device Functions	7-97
Creating and Manipulating View Device Objects	7-97
GXNewViewDevice	7-98
GXDisposeViewDevice	7-99
GXCopyToViewDevice	7-100
GXEqualViewDevice	7-101
Manipulating View Device Object Properties	7-102
GXGetViewDeviceClip	7-102
GXSetViewDeviceClip	7-103
GXGetViewDeviceMapping	7-105
GXSetViewDeviceMapping	7-106
GXGetViewDeviceBitmap	7-107
GXSetViewDeviceBitmap	7-108
GXGetViewDeviceViewGroup	7-109
GXSetViewDeviceViewGroup	7-109
GXGetViewDeviceAttributes	7-110
GXSetViewDeviceAttributes	7-111
GXGetViewDeviceTags	7-112
GXSetViewDeviceTags	7-113
Retrieving the View Devices That Intersect a Shape	7-115
GXGetShapeGlobalViewDevices	7-115
Measuring a Shape in Device Coordinates	7-116
GXGetShapeDeviceBounds	7-116
GXGetShapeDeviceArea	7-118
Measuring the Colors and Pattern Width of a Shape on a Device	7-118
GXGetShapeDeviceColors	7-119
Hit-Testing a Shape on a Device	7-120
GXHitTestDevice	7-120

View Group Functions	7-121
Creating and Disposing of View Group Objects	7-121
GXNewViewGroup	7-122
GXDisposeViewGroup	7-122
Getting the View Ports and View Devices of a View Group	7-123
GXGetViewGroupViewPorts	7-123
GXGetViewGroupViewDevices	7-124
Measuring a Shape in Global Coordinates	7-125
GXGetShapeGlobalBounds	7-125
Summary of View-Related Objects	7-127
Constants and Data Types	7-127
View Port Functions	7-129
View Device Functions	7-130
View Group Functions	7-131

This chapter describes view-related objects—view ports, view devices, and view groups—and the functions you can use to manipulate them. Read this chapter to learn how to draw any of the QuickDraw GX shapes you create, and how to control the representations of those shapes on any output devices.

Before reading this chapter, you should be familiar with the information in the chapter “Introduction to QuickDraw GX” in this book. You should also be familiar with shape objects, as discussed in the chapter “Shape Objects” in this book, and transform objects, as discussed in the chapter “Transform Objects.” Additional information related to color drawing is found in the chapter “Color and Color-Related Objects.”

This chapter constitutes the complete discussion of view-related objects for QuickDraw GX. Unlike for shape objects and style objects, there is no additional discussion of view ports, view devices, or view groups for graphic or typographic shapes in other books. The book *Inside Macintosh: QuickDraw GX Environment and Utilities* does, however, discuss drawing to Macintosh windows and how to relate a view port to a window. It also discusses matrix manipulation, which you can use to change the mapping property of a view port or view device.

This chapter introduces view ports, view devices, and view groups, discusses their properties, and shows how they are related to each other. It then discusses the different coordinate spaces you use to manipulate these objects. It then shows how to use QuickDraw GX functions to

- n use view ports: create and manipulate them; manipulate their properties, including clip, mapping, dither, halftone, and parent and child view ports; and analyze a shape in a view port
- n use view devices: create and manipulate them; manipulate their properties, including clip and mapping; and analyze a shape on a view device
- n use view groups: create and manipulate them; set them up for offscreen drawing; and analyze a shape in a view group

About View Ports, View Devices, and View Groups

The view-related objects in QuickDraw GX exist to support drawing. They work together to provide device-independent drawing destinations for an application, while at the same time allowing access to device characteristics and permitting drawing destinations and devices to be grouped and manipulated in flexible ways.

These are the view-related objects:

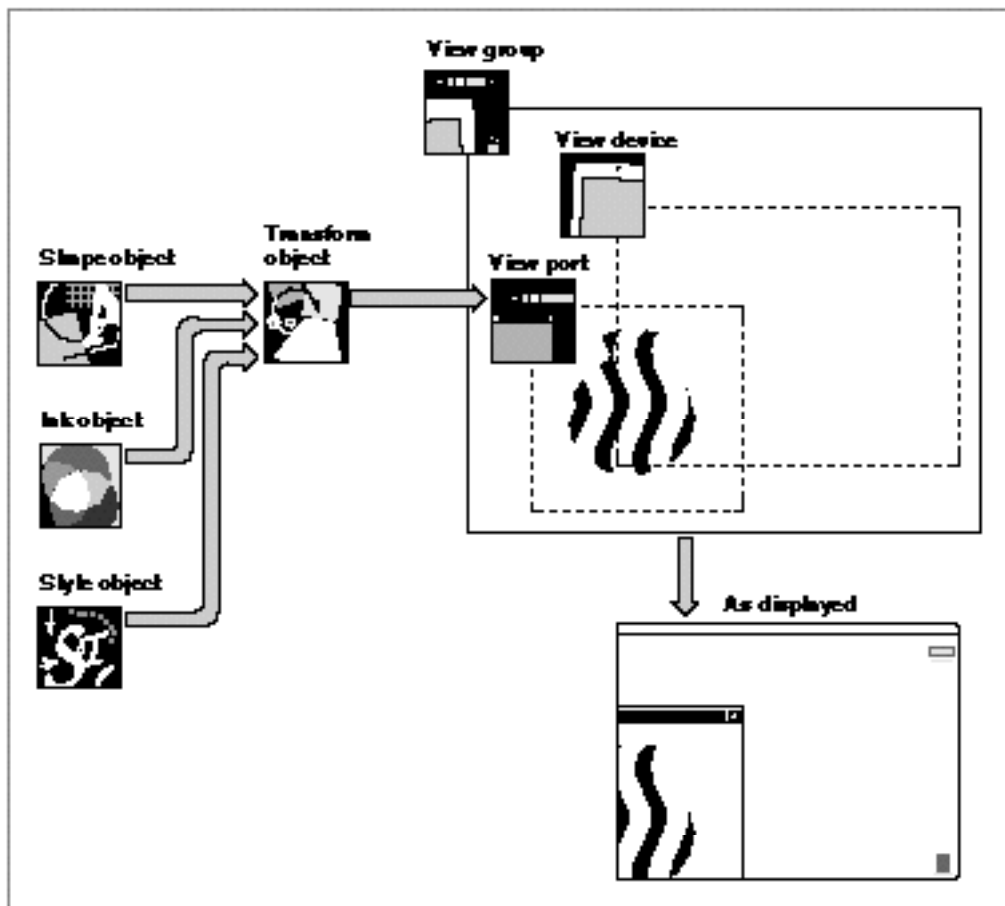
- n **View port.** A view port represents a drawing destination, such as the content area within a window. View port objects are device-independent, and have mapping and clipping properties that position, modify, and mask the shapes drawn into them. A shape’s transform object defines the view ports that the shape is drawn into.

View-Related Objects

- n **View device.** A view device represents a physical device, such as a monitor or printer, on which objects are drawn. Each view device object has mapping and clipping properties that define its resolution, mask its visible area, and position it in relation to view ports and other view devices.
- n **View group.** A view group relates view ports and view devices to each other. Each view group object identifies a particular set of view ports and view devices; it also defines a coordinate space that positions view ports and view devices relative to each other. Drawing occurs in those locations where view ports and view devices within the same view group overlap.

Figure 7-1 shows the relationships among the view-related objects and the sequence of events that occur when a shape is drawn.

Figure 7-1 Objects used by the drawing mechanism



As Figure 7-1 shows, a shape's geometry, initially modified by information contained in the shape's style object and ink object, is further modified by the clip and the mapping of the transform object. That modified shape is then even further modified by the mapping and clip of one or more view ports, and modified once more by the mapping and clip of any view devices that intersect the view ports.

A shape cannot be drawn unless its transform object contains a reference to at least one view port. The transform object for the shape in Figure 7-1 references only one view port, so the shape is drawn only once. Transform objects are described in the chapter "Transform Objects" in this book.

Drawing occurs where the clip of a view port overlaps the clip of a view device within the same view group. The overlap is determined by the dimensions of the two clip shapes, both of which are defined in terms of the view group's coordinate space. In this example, the position of the shape in the view port and the overlap between the view port and the view device mean that only part of the shape is rendered.

The view group in Figure 7-1 represents a coordinate space for all view ports and view devices that may be visible to the user of your application. This particular view group is called the **onscreen view group** because the view devices represent actual screen devices. You can create view groups to draw offscreen as well.

View-related objects are different from most other QuickDraw GX objects in several ways:

- n All the view-related objects can be shared, but they have no owner count and cannot be cloned. When you dispose of a view-related object, it is deleted from memory.
- n View ports can be arranged hierarchically, and you can attach a view port hierarchy to a Macintosh window to simplify clipping, moving, and scrolling documents in a window.
- n QuickDraw GX creates view device objects for all physical screen devices for you; for drawing to the screen, you normally never need to create a new view device.

About View Port Objects

A **view port** object represents the drawing destination for QuickDraw GX objects. A view port is analogous in some ways to a porthole on a ship, hence the name *port*. Objects seen through the porthole may have any extent, but only those parts within the boundaries of the porthole are visible.

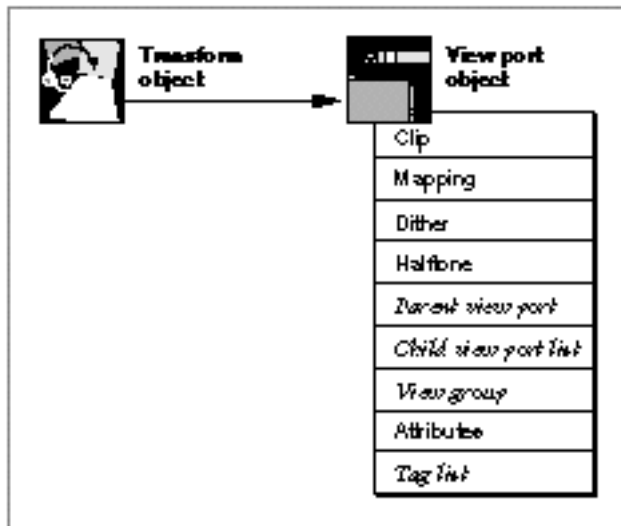
View ports are device independent, and you normally need not take device characteristics such as pixel resolution into account when you draw.

View Port Properties

The interface to view port objects is entirely procedural. You manipulate the information in a view port by modifying its properties using QuickDraw GX functions.

View port objects have nine accessible properties, as shown in Figure 7-2. Note that, because a view port is an object and not a data structure, the order of the properties as shown in Figure 7-2 is completely arbitrary. Properties in italics are references to other objects.

Figure 7-2 View port object properties



These are the accessible properties of a view port:

- n **Clip.** A specialized shape geometry that controls the visibility of all shapes drawn to this view port. Only the parts of shapes that overlap with the clip remain visible when they are drawn. In the porthole analogy for a view port, the view port clip represents the transparent area of the porthole. The view port clip is further described in the next section, “View Port Clip and Mapping.”
- n **Mapping.** A mathematical matrix that specifies the translation, scaling, skewing, rotation, and perspective of all shapes drawn to this view port. The view port mapping is further described in the next section, “View Port Clip and Mapping.”
- n **Dither.** A value that specifies the dither setting of a view port. Dithering combines pixels of different colors to create the illusion of more colors than are actually supported by the hardware of an output device. For more information about the dither property, see the section “Dither” beginning on page 7-10.
- n **Halftone.** A structure that specifies the halftone settings of the view port. Halftones use variable-sized dots of color to create the illusion of more colors than are actually supported by the hardware of an output device. For more information about the halftone property, see the section “Halftone” beginning on page 7-13.
- n **Parent view port.** A reference to the view port that is the parent of this one. View ports exist in a hierarchical relationship that simplifies attaching them to windows and moving groups of them as units. For more information about the parent view port and view port hierarchies, see the section “Parent and Child View Ports” on page 7-18.

View-Related Objects

- n **Child view port list.** A list of references to the view ports for which this view port is the immediate parent. View ports exist in a hierarchical relationship that simplifies attaching them to windows and moving groups of them as units. For more information about child view ports and view port hierarchies, see the section “Parent and Child View Ports” on page 7-18.
- n **View group.** A reference to the view group object to which this view port belongs. View groups are described in the section “About View Group Objects” beginning on page 7-29.
- n **Attributes.** A set of flags that affect various characteristics of the shapes drawn to this view port. See the section “View Port Attributes” on page 7-20 for more information.
- n **Tag list.** A list of references to custom information about this view port object, stored in private structures called tag objects. The chapter “Tag Objects” in this book describes tag objects in general and how you can use them to add custom information to objects.

QuickDraw GX provides functions to manipulate each of these view port object properties.

View Port Clip and Mapping

Like transform objects, view port objects have a clip property and a mapping property. A view port’s mapping and clip are applied to a shape after the transform clip and mapping have already been applied.

The clip and mapping properties for a view port follow the same general conventions as for transform objects. The clip property specifies a shape geometry that you use as a mask to restrict the visibility of a shape object when it is displayed or printed. The clip is equivalent to a primitive shape, a shape whose geometry and fill properties by themselves define the shape. Specifically, a clip can be a framed or filled geometric shape, a glyph shape, a 1-bit-per-pixel bitmap shape, or an empty or full shape. Primitive shapes are described in more detail in the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*.

The filled or framed parts of a view port clip define the areas in which a shape drawn to that view port show through. If the clip shape is a filled rectangle, for example, only the parts of a shape that are within the limits of that rectangle are visible. Commonly, view port clips are filled rectangles, because the visible parts of view ports commonly correspond to rectangular windows or panes of windows.

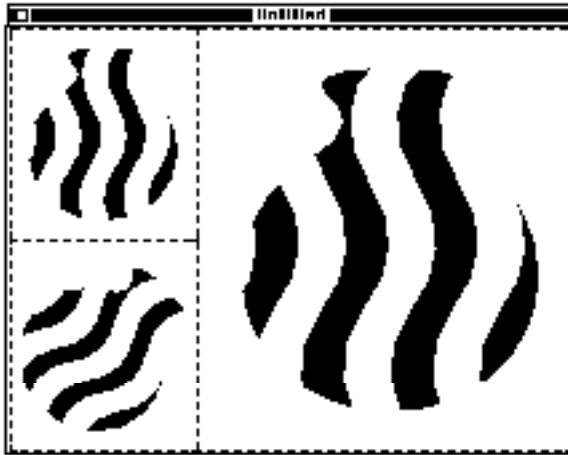
The mapping property of a view port is a 3×3 matrix that specifies one or more transformations that a view port applies to all shapes drawn into it. You can use the view port mapping to perform operations such as the following:

- n translation, which changes the positions of shapes in the view port
- n scaling, which shrinks or enlarges shapes horizontally or vertically or both
- n rotation, which turns shapes about a fixed point
- n skewing, which distorts shapes progressively along a single axis
- n perspective, which distorts shapes to provide a three-dimensional effect

Most commonly, you use the view port mapping to position the view port (equivalent to positioning a window), and possibly to scale or rotate its contents. Skewing and perspective are less common, but you can use them for special effects. You can specify the identity mapping, a matrix whose elements have the value 1.0 along the diagonal and 0.0 elsewhere, to leave shapes drawn to this view port unchanged from the application of the transform mapping.

Figure 7-3 shows three different view ports in a single window, all used to display the same shape as that shown in Figure 7-1 on page 7-6. The shape is a vase, and its transform clip causes it to appear as wavy stripes.

Figure 7-3 Clipping and mapping in view ports



In Figure 7-3, each view port's clip shape is a rectangle that defines its pane in the window. The upper left view port uses an identity mapping. The lower left view port's mapping specifies a clockwise rotation. The right view port's mapping specifies scaling (equal in x and y dimensions) that enlarges the shape.

Dither

Dithering is a technique of assigning alternating colors to adjacent pairs or groups of pixels in a device's bitmap to achieve the illusion of a color that cannot be represented directly. For example, if a device only supports three shades of blue, dithering allows QuickDraw GX to assign those colors in a specific order to adjacent pixels, so that the mix of shades in the combined pixels approximates a desired but unsupported shade.

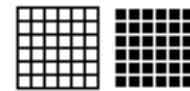

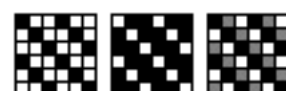

Dithering works one way in shapes that have a uniform color, and another way in bitmaps.

Dithering of Shapes Other Than Bitmaps

The dither property of a view port specifies a **dither level**, which is the maximum number of colors that QuickDraw GX can use in dithering when it draws a shape. The dither level can be between 1 and 16; a dither level of 1, the simplest, is equivalent to no dithering. A level of 0 is not permitted. Dithering has no effect if the device resolution is 32 bits per pixel.

Table 7-1 shows the pixel patterns that can occur, depending on the dither level. A dither level of 1 provides a solid pattern, which is effectively no dithering. A dither level of 2 provides a 2-by-2 repeating checkerboard pattern. A dither level of 3 provides a 3-by-3 repeating stripe pattern. A dither level of 4 provides a 4-by-2 repeating pattern. Note that the effective resolution of an image decreases as dither level increases, because each set of pixels that make up a unit of the dither pattern function as a single, larger pixel of dithered color.

Table 7-1 Dither levels and patterns

Dither level		Available patterns
1		
2	Above patterns, plus:	
3	Above patterns, plus:	
4	Above patterns, plus:	

Implementation Note

Version 1.0 of QuickDraw GX supports a maximum of 16 colors in a dither pattern, although 4 is the practical maximum dither level, especially for grayscale drawing. u

QuickDraw GX does not necessarily use exactly the dither pattern specified by the dither property, unless the ink object attached to the shape drawn to the view port has its `gxForceDitherInk` attribute set. For example, if you specify a dither level of 4, QuickDraw GX may use any pattern from level 1 through level 4, as necessary, to create the illusion of additional colors. If the ink object's `gxForceDitherInk` attribute is set, however, only the level-4 pattern is used. Conversely, if the ink object's `gxSuppressDitherInk` attribute is set, no dithering occurs. Note also that you can affect the pixel alignment of the dither pattern with the ink object's `gxPortAlignDitherInk` attribute. For more information about these ink attributes, see the chapter "Ink Objects" in this book.

QuickDraw GX uses information from the color profile for the view device object to determine the supported colors and it chooses the appropriate colors to use in the dither for you. Although the results of dithering are controlled by the view device, you specify the dither level in the view port object so that you can simulate its effect, on the computer that is running the application, for a device that may not actually be present.

Dithering of Bitmaps

When you draw a bitmap shape, dithering works differently. Unlike with single-color shapes, dithering of bitmaps uses no specific pattern and recognizes no different dither levels. Dithering is off if the dither level is 1, and it is on if the dither level is greater than one.

Dithering of bitmaps uses the process of **error diffusion**, in which the error (the difference between the computed color of a given pixel and the nearest color available on the view device) is passed to adjacent pixels. The dithering algorithm starts at the top left of the visible part of the bitmap, and progresses through the bitmap, traveling left to right across one row of pixels and then right to left across the next lower row, and so on until the entire bitmap has been traversed. For each pixel, the algorithm adds the accumulated error (passed from the pixel above it and the pixel to the left of it) to the computed color, picks the closest available device color for that pixel, and passes the new error (the new computed color minus the available color) to the pixel to the right and the pixel below; half of the error goes to each.

Not all dither-related ink attributes are applicable to dithering of bitmaps. Setting or clearing the `gxPortAlignDitherInk` attribute or the `gxForceDitherInk` attribute of the ink object attached to a bitmap shape has no effect. Setting the `gxSuppressDitherInk` attribute, however, does have the effect of turning off dithering, even for bitmaps.

Ink attributes are described in the chapter "Ink Objects" in this book. The bitmap shapes chapter of *Inside Macintosh: QuickDraw GX Graphics* shows an example of drawing a dithered bitmap.

Drawbacks of Dithering

Dithering can provide good results in many cases, but it does have drawbacks:

- n Dithering slows down the drawing operation slightly for non-bitmap shapes, because of the overhead of determining the pattern in which colors are assigned. Dithering significantly slows down the drawing of bitmaps.
- n You cannot reliably use an ink transfer mode to reverse the effect of drawing a color that is dithered. Thus, transfer modes such as highlight mode may not be completely reversible where dithering occurs.
- n In bitmap dithering, because QuickDraw GX determines which color to apply to a pixel using the color of the pixel next to it, a cumulative dithering error can occur due to the way a preceding pixel is changed by the dithering algorithm.
- n Because clipping can interrupt the error diffusion in a bitmap dither, dithering a clipped shape can produce a different result from dithering the same unclipped shape. It can also mean that redrawing portions of an image, as when scrolling, can cause seams, or visible lines, between the separately drawn portions.

Note that dithering and halftones (described next) are mutually exclusive; if you choose both simultaneously, only a halftone is used.

Halftone

A **halftone** is a pattern of alternating colors of variable intensities in a fixed cell size, used to represent a variety of colors. Halftoning, like dithering, provides a method of representing color by alternating the available colors on view devices that support only a limited number of colors. Unlike a dither pattern, however, a halftone's fixed cell size means that its resolution is constant and adjacent halftones representing different colors mesh well. Also, unlike with dithers, you specify the colors that make up a halftone. If you use halftones, you should be familiar with how QuickDraw GX represents color, as described in the chapter "Color and Color-Related Objects" in this book. Also, note that dithering and halftones are mutually exclusive; if you choose both simultaneously, only a halftone is used.

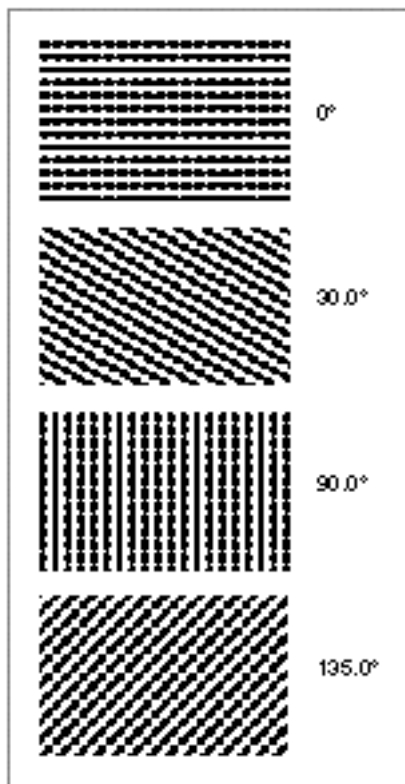
A halftone consists of a pattern of variable-sized dots of one color against a background of another color. The halftone simulates a desired color (such as a specific intensity of gray), with the proper proportion of dot color (such as black) and background color (such as white). The colors are not limited to single components, however; a halftone can simulate beige, for example, by mixing pink with yellow. QuickDraw GX can attempt to reproduce any desired color by mixing the dot color and background color specified in a halftone. You can specify any color as the background and any other color as the dot color, and QuickDraw GX will find the best mixing proportion, given the specifications that you provide. Commonly, however, if you are using halftones you work with a single color component (such as blue for RGB space or yellow for CMYK space), and you specify that component as the dot color and black (for RGB) or white (for CMYK) as the background color.

A halftone is described by several characteristics, specified in the `gxHalftone` structure:

```
struct gxHalftone{
    Fixed          angle;           /* direction of halftone */
    Fixed          frequency;       /* cells per inch */
    gxDotType     method;          /* kind of pattern */
    gxTintType    tinting;         /* tint calculation method*/
    gxColor       dotColor;        /* color of dots */
    gxColor       backgroundColor; /* color of background */
    gxColorSpace  tintSpace;       /* color space for tint */
};
```

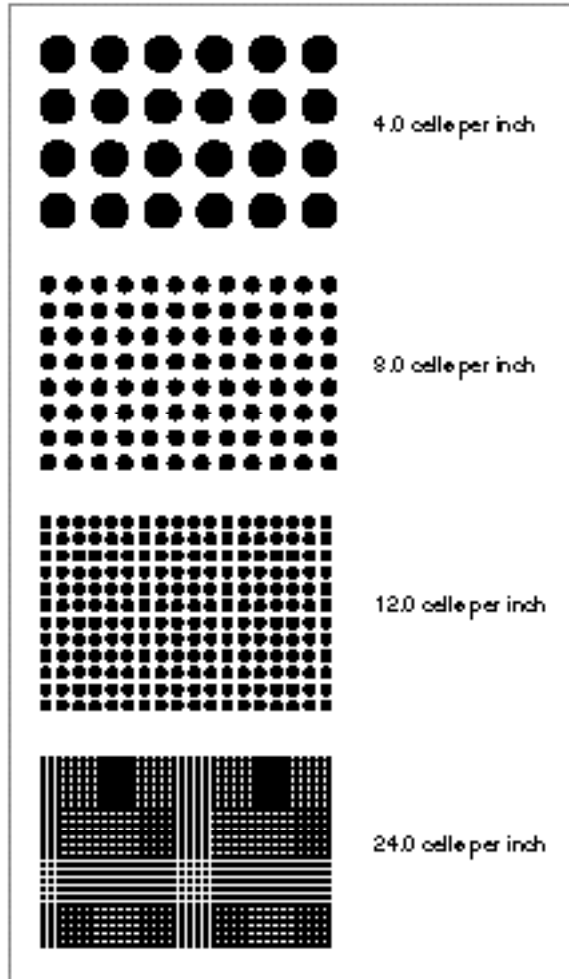
The **angle** describes the orientation of the rows of dots in the halftone pattern. It is a fixed-point number between 0.0 and 360.0 that describes an angle, in degrees, clockwise from horizontal. Figure 7-4 shows several angles.

Figure 7-4 Halftone angle



Each cell in a halftone is an area that contains some proportion of background color and dot color. The **frequency** describes the size of the cells, in terms of numbers of dots per inch. You typically specify a frequency based on desired output quality and device resolution. Figure 7-5 shows examples of various frequencies.

Figure 7-5 Halftone frequency



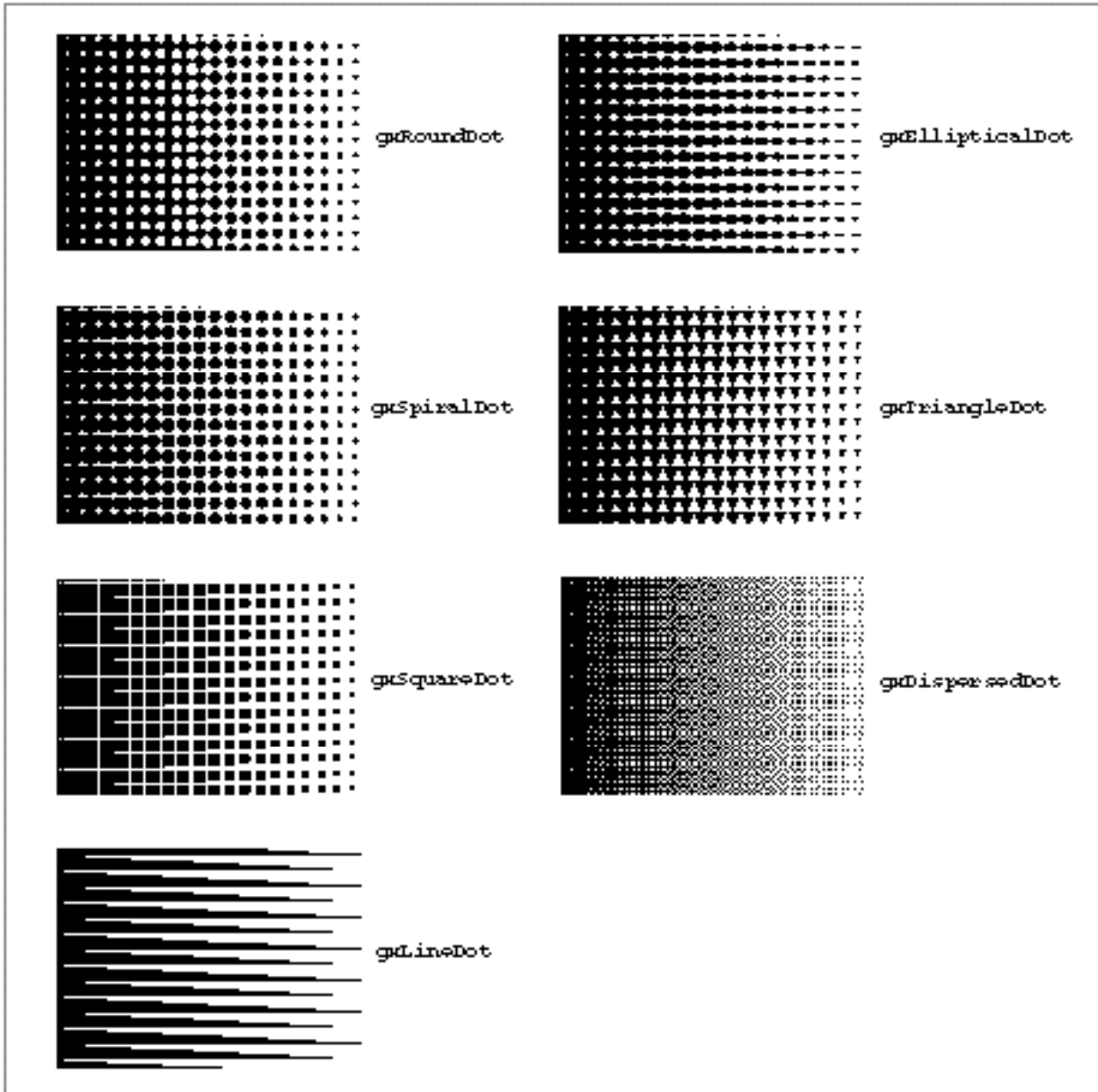
The **method**, or **dot type**, describes the halftone pattern itself and how it is filled: the shapes of the dots, the pattern of their arrangement, and the way in which a dot fills its cell as it enlarges. The supported methods are defined in the `gxDotTypes` enumeration:

```
enum gxDotTypes{
    gxRoundDot = 1,
    gxSpiralDot,
    gxSquareDot,
    gxLineDot,
    gxEllipticDot,
    gxTriangleDot,
    gxDispersedDot
};

typedef long gxDotType;
```

Figure 7-6 shows examples of these patterns.

Figure 7-6 Halftone dot types



The tinting, or **tint type**, specifies how the input color (the original color to which the halftone is applied) is to be approximated by a ratio of dot color and background color. The **tint** is a calculated value from zero to one: if the tint is zero, the halftone is composed only of the background color (the dots are infinitesimally small); if the tint is one, the halftone is composed entirely of the dot color (the dots fill their cells entirely).

View-Related Objects

The **tint color** is the actual color represented by the combination of dot and background, and is therefore a weighted average of the dot color and background color. The tint color may be only an approximation to—or even just a single component of—the input color.

The `gxTintTypes` enumeration, described on page 7-67, defines these tinting choices:

- n **Luminance.** The tint color is the input color's luminance. The gray closest to the luminance of the input color is used to calculate the tint value. This tinting method is used for making grayscale halftones from grayscale or color images.
- n **Color component.** The tint color is some intensity of any one of the components of the input color. This tinting method is used for making halftoned color separations.
- n **Color average.** The tint color is the average of the components of the input color, determined by adding up the color components and dividing them by the number of components. This tint type is used with RGB only, and follows this formula:

$$\text{tint} = 1 - (R + G + B) / 3$$

Color average is different from luminance in that, for example, the luminance of an RGB color is not exactly the average of its component intensities. This tinting method is used for making grayscale halftones from color images.

- n **Color mixture.** The tint color is a point on the line (in color space) connecting the dot color and the background color. The orthogonal projection of the input color onto that line locates the tint color, and the tint (the proportion of dot to background) is defined by the position of that projection point on the line. This is the formula:

$$\text{tint} = 1 - D1 / (D1 + D2)$$

where

D1 = input color-dot color distance

D2 = input color-background color distance

This tinting method is used for getting the closest possible representation of any input color using any dot and background colors.

The **dot color** and **background color** are, respectively, the color of the dots and the color of the background used to form the halftone. In the halftone structure, they are full color specifications, not just single color-component values. In setting up a halftone structure, you need to specify these colors in a way that is meaningful considering the tint type you have chosen. For example, if you are creating a halftoned color separation, you typically use a dot color that is the same color as the color component specified in the tint type, and a background color of white.

The **tint space** describes the color space the input color is converted to before the tint value is determined. For instance, you can set the tint space to CMYK space to separate out the cyan portion of an image that may have been created in RGB space. It is not necessary for the input colors or the view device colors to be set to CMYK space, only the halftone.

Note that halftoning occurs for a shape only if its view port contains a valid halftone structure, and if its ink object's `gxSuppressHalftoneInk` attribute is cleared. The `gxSuppressHalftoneInk` attribute is described in the chapter "Ink Objects" in this book.

Parent and Child View Ports

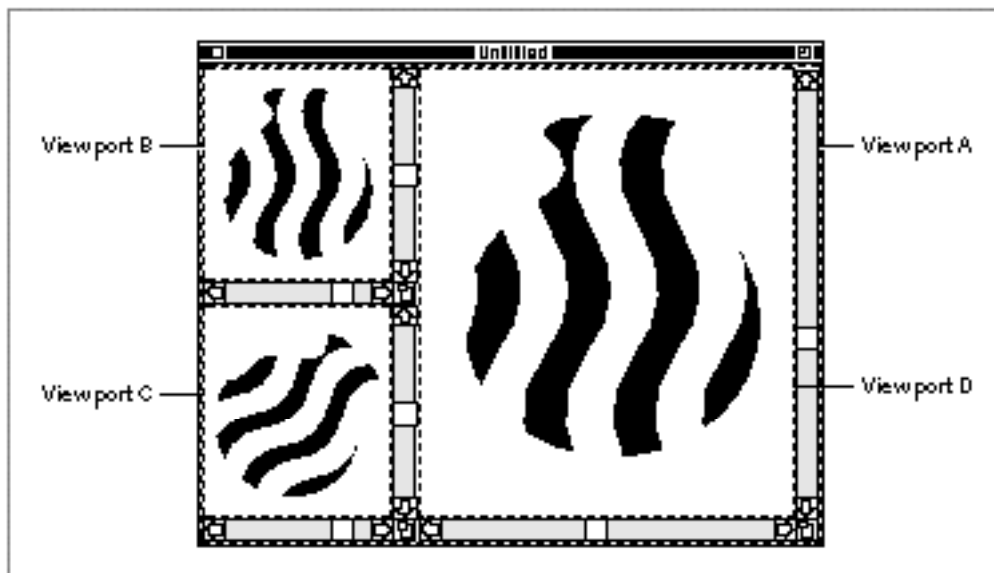
Two view port properties, the parent view port and the child view port list, allow you to arrange view ports in a hierarchy. The primary advantage of this capability is that QuickDraw GX manages the positional relationships among several related view ports for you.

In a view port object, the hierarchical relationship is represented by parent and child view port references. Each view port can reference one parent view port and any number of child view ports. When you move a view port by altering its clip and mapping, QuickDraw GX moves all its child view ports (and their child view ports, if any) accordingly. If the parent view port of your view port is attached to a window, QuickDraw GX moves your view port (and its children) to match movements of the parent whenever the user moves the window.

A view port hierarchy consists of a root view port, which is one with no parent view port, and all of its child view ports. If a child view port is also a parent view port, its children are part of the hierarchy too, and so on. Any parent view port in a hierarchy also defines a subhierarchy that consists of itself as the root, its child view ports, their child view ports, and so on.

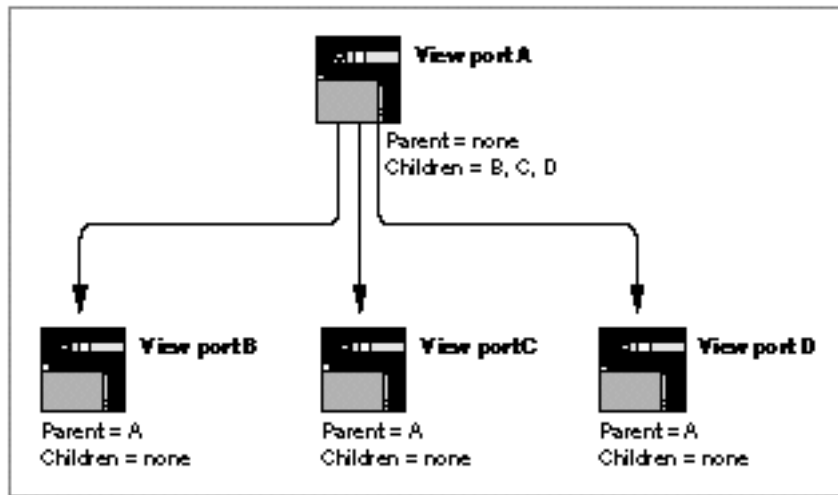
Consider, for example, the window shown in Figure 7-7. Like in Figure 7-3 on page 7-10, it displays three different views of a vase. In this case, however, all the views are scrollable, which requires four view ports in a hierarchy.

Figure 7-7 Hierarchical view ports in a window



The four view ports associated with the window in Figure 7-7 are arranged in the simple hierarchy shown in Figure 7-8.

Figure 7-8 A view port hierarchy



View port A encompasses the entire content area of the window. It does not have a parent, so it is the root of the hierarchy. View port A has three child view ports: B, C, and D. View ports B, C, and D all have the same parent, view port A. None of them, however, have child view ports of their own.

This hierarchical organization allows QuickDraw GX to automatically move all view ports when the window is moved. It also allows you to support scrolling in view port B, C, or D with minimal effort. When the user scrolls pane B, for example, the translation in view port B's mapping is changed to reflect the shape's new position in the window. No changes are required to the other child view ports or to view port A to implement scrolling. See the section "View Port Objects and Windows" beginning on page 7-21 for more information on view port hierarchies and windows.

When you set up a view port hierarchy, you create the root view port by calling the `GXNewWindowViewPort` function if you want the view port to be associated with a window, or the `GXNewViewPort` function otherwise. You create child view ports by calling the `GXNewViewPort` function for each, and then using the `GXSetViewPortParent` and `GXSetViewPortChildren` functions to organize them into a hierarchy.

The following rules apply when you set up a view port hierarchy:

- n You cannot create a circular relationship among view ports. For example, a parent view port cannot also be a child view port within its own hierarchy.
- n The view ports in a hierarchy must all be in the same view group.

View Port Attributes

Each view port object has a set of attributes, a group of flags that specify different aspects of display behavior. View port attributes allow you to specify drawing in gray only, to constrain shapes to integral pixel locations, or to enable color matching for shapes drawn to the port. Table 7-2 lists the constants for the view port attribute and describes what each one means. The constants are defined in the `gxPortAttributes` enumeration.

Table 7-2 View port attributes

Constant	Value	Explanation
<code>gxGrayPort</code>	<code>0x0001</code>	If set, QuickDraw GX only allows grays to be drawn to the view port; it converts colors into a gray color space. Color spaces are described in the chapter “Color and Color-Related Objects” in this book.
<code>gxAlwaysGridPort</code>	<code>0x0002</code>	If set, QuickDraw GX sets the <code>gxDeviceGridStyle</code> style attribute for all shapes drawn to the view port. This has the effect of constraining a shape to integral pixel values, thus avoiding distortion due to rounding of fractional coordinates. For more information about the <code>gxDeviceGridStyle</code> attribute, see the geometric styles chapter of <i>Inside Macintosh: QuickDraw GX Graphics</i> .
<code>gxEnableMatchPort</code>	<code>0x0004</code>	If set, QuickDraw GX performs color matching for all shapes drawn to this view port. Note that you must set this attribute for color matching to occur; color matching is off by default in view ports. Color matching is described in the chapter “Color and Color-Related Objects” in this book.

The Default View Port Object

When you first create a view port object, you must assign it to a specific view group. Other than that, the view port has these default properties:

- n No parent view port.
- n An empty child view port list.
- n A clip shape that is a full shape. The clip has no effect.
- n A mapping that is the identity mapping. The mapping has no effect.

View-Related Objects

- n A dither level of 1.
- n A `nil` halftone (no halftone).
- n No attributes set.
- n An empty tag list.

When you create a transform object, or if you just use the original default shape when you create a shape object, QuickDraw GX uses the default transform and assigns a default view port to the transform's view port list. That view port is in the onscreen view group and has the default properties just listed. That means that you can simply create a shape object and immediately draw it to the screen, without creating any view port. However, for most application purposes you need to restrict drawing to the interiors of windows. To do that, you can create a view port each time the user opens a window, and then alter the default shape object for each shape type to make sure that it references a transform whose view port list includes that view port or a child view port of it. Alternatively, you can explicitly assign the proper view port to the transform of each shape after the shape is created.

For more information about the onscreen view group, see "Onscreen and Offscreen View Groups" on page 7-29.

View Port Objects and Windows

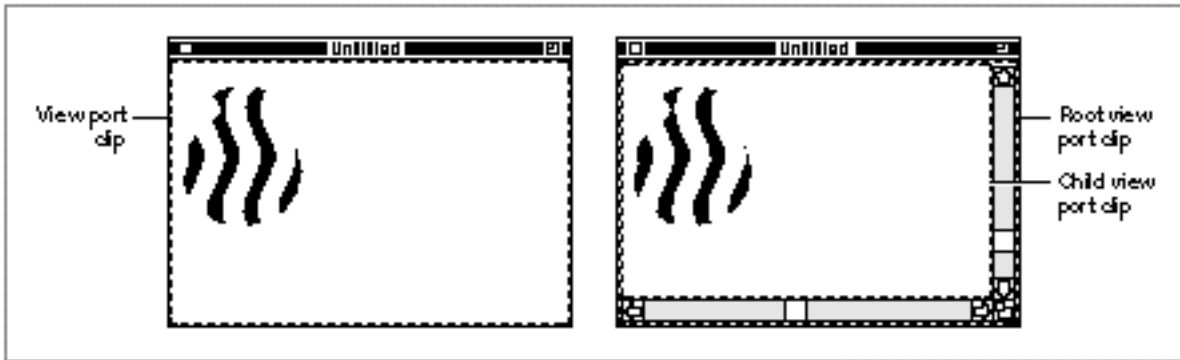
In most cases on the Macintosh, when your application draws to the screen it draws into a Macintosh window. (You do not need a window for offscreen drawing or when printing.) Most of the view ports you create, therefore, are in some way associated with windows. QuickDraw GX allows you to associate a view port with a window, tying it to the window and establishing it as the root of a view port hierarchy.

To attach a view port to a window, call the `GXNewWindowViewPort` function. This function sets up the view port so that drawing occurs only in the content area of the window (everything except for the title bar), effectively as if the view port's clip were equal to the window's visible region. When the user moves the window or changes its size, QuickDraw GX automatically moves the view port and adjusts its drawing limits to match the visible region. QuickDraw GX does not allow you to modify the clip or mapping of that view port.

If you add child view ports to the view port hierarchy, they are also moved as the window is moved. However, if the window is changed in shape, you need to adjust the clips of the child view ports to coincide with the new window dimensions.

Figure 7-9 shows a shape object drawn in two windows. In the window on the left, the shape is drawn directly to the window's view port; in the window on the right, the shape is drawn to a child view port of the window's view port.

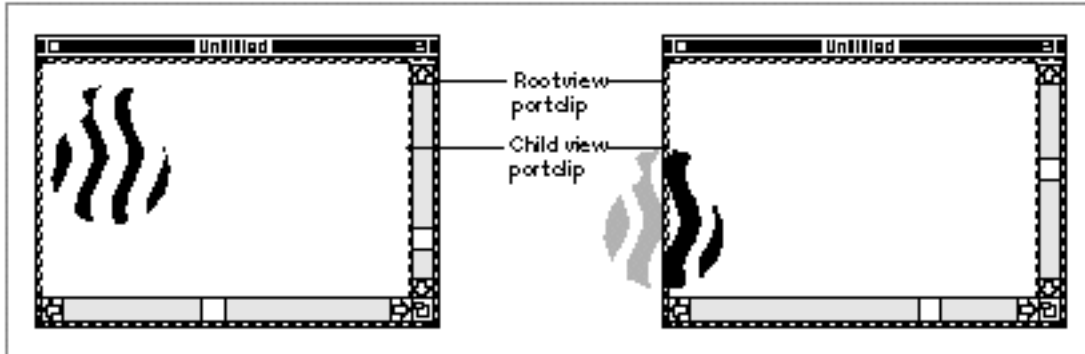
Figure 7-9 View ports in windows



One reason to draw only into a child view port is that it facilitates drawing tasks such as scrolling. Using a child view port helps to separate window management from content management when drawing. You use the parent view port for tracking window movement and visibility, and you manipulate the child view port's properties directly, without concern for the position or visibility of the parent view port. To implement scrolling, for example, you can follow these steps:

1. Create a child view port for your window's view port.
2. Draw your shapes to the child view port.
3. Alter the child view port's mapping to reflect the translation caused by scrolling.

Figure 7-10 shows these steps schematically. Note that the scroll bars are part of the content area of the window, and adjusting them means drawing into the parent view port. Note also that the child view port clip is smaller than the content area of the window, so that drawing into it does not draw over the scroll bars.

Figure 7-10 Adjusting a child view port's mapping to handle scrolling

You need not adjust the child view port's clip after scrolling because the clip's position is not changed when the mapping is altered; you need to adjust the clip only when the dimensions of the child view port's drawing area are changed (such as when the window is resized). Remember also that you need to adjust the mapping of a child view port only when there is relative movement between the child view port and its parent; if the user simply moves the window, you do not need to adjust the child view port because QuickDraw GX handles this for you.

For information about how clipping and mapping interact, see the section “About Drawing, Coordinate Conversion, and Clipping” beginning on page 7-30.

For information about the `GXNewWindowViewPort` function, see the environment chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*. For an example that uses this function, see page 7-41.

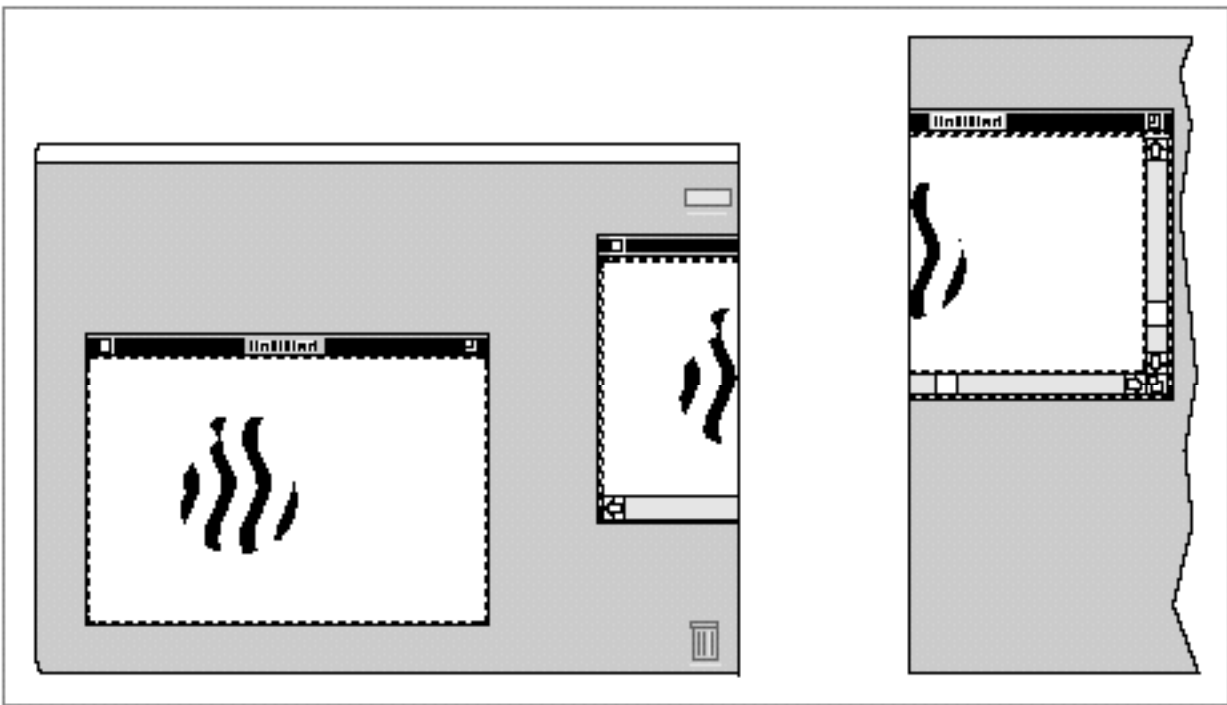
Drawing is Not Restricted to Windows

QuickDraw GX view ports are not restricted to windows; you can draw anywhere in a coordinate space. This feature makes it easy to take complete control of a view device and draw anywhere on it. For example, you can support dragging of objects between windows in this way. On the other hand, because most QuickDraw GX applications must share a view device with windows from other applications, you typically want to restrict drawing to window content areas. u

About View Device Objects

A **view device** object represents an output device such as a monitor or printer. When a shape is drawn, it appears on a view device, although its actual drawing destination is a view port. The intersection of the view port and view device determine where and how much of the shape is drawn. Figure 7-11 shows how a single view device can display more than one view port, and how a single view port can overlap more than one view device.

Figure 7-11 View ports overlapping view devices

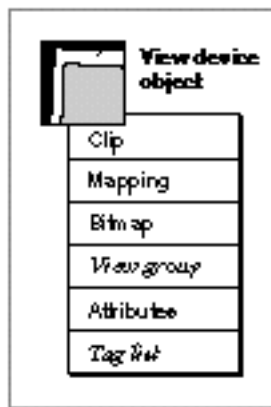


For drawing, you do not need to be concerned whether a view port overlaps one or more separate view devices; you just draw to the view port and QuickDraw GX handles it for you. On the other hand, if a view port does not intersect any device, the shapes drawn to the view port are not rendered at all.

View Device Properties

View device objects have six accessible properties, as shown in Figure 7-12. Note that, because a view device is an object and not a data structure, the order of the properties as shown in Figure 7-12 is completely arbitrary. Properties in italics are references to other objects.

Figure 7-12 View device object properties



These are the accessible properties:

- n **Clip.** A specialized shape geometry that defines the active imaging area of the view device. Only the parts of the view device's bitmap that overlap with the clip can be drawn to. The view device clip is further described in the next section, "View Device Clip and Mapping."
- n **Mapping.** A mathematical matrix that specifies the translation, scaling, rotation, skewing, and perspective of shapes drawn on this view device. The view device mapping is further described in the next section, "View Device Clip and Mapping."
- n **Bitmap.** A bitmap structure that represents the imaging area of the view device. The view device bitmap is further described in the section "View Device Bitmap" on page 7-26.
- n **View group.** A reference to the view group object to which this view device belongs. View groups are described in the section "About View Group Objects" beginning on page 7-29.
- n **Attributes.** A set of flags that affect the state of activity and memory use of this view device. See the section "View Device Attributes" on page 7-27 for more information.
- n **Tag list.** A list of references to custom information about this view device object, stored in private structures called tag objects. The chapter "Tag Objects" in this book describes tag objects in general and how you can use them to add custom information to objects.

Note that QuickDraw GX sets the properties for all onscreen view devices (view device objects that represent physical display devices present on the user's system). You cannot change the properties of those view devices.

View Device Clip and Mapping

Like transforms and view ports, view device objects have a clip property and a mapping property. A view device's mapping and clip are applied to a shape after those of the transform and the view port have already been applied.

The clip and mapping properties for a view device follow the same general conventions as for transform objects. The clip property specifies a mask that restricts the area on the device in which drawing or printing takes place. The clip is equivalent to a primitive shape, a shape whose geometry and fill properties by themselves define the shape. Specifically, a clip can be a framed or filled geometric shape, a glyph shape, a 1-bit-per-pixel bitmap shape, or an empty or full shape. Primitive shapes are described in more detail in the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*.

The filled or framed parts of the clip define the areas in which drawing can occur. In most cases the view device clip is simply a filled rectangle, often covering exactly the imageable area of the device. You can, however, restrict drawing to a single portion of the device by making the clip shape smaller.

The mapping property of a view device is a 3×3 matrix that specifies one or more transformations that the view device applies to shapes drawn into it. You can use the view device mapping like other mappings, to perform translation, scaling, rotation, skewing, or perspective. However, in most cases the view device mapping is used only to position the view device relative to other view devices and view ports, and to define its pixel size (an identity mapping usually means that pixel size is 72 per inch). You normally do not need to modify a view device's mapping, although it is possible for view device objects that you create yourself.

View Device Bitmap

The bitmap property of a view device is stored as a bitmap structure (type `gxBitmap`) that represents the imaging area of the device. The bitmap specifies the height, width, and pixel depth of the view device. The upper left corner of the pixel image is the upper left corner of the imaging area of the device; if the view device object has an identity mapping, that also corresponds to location (0.0, 0.0) in the view group to which the view device belongs.

The bitmap also specifies, possibly by using a reference to a color set object, the color of each pixel and the set of available colors on the device. (Bitmaps with fewer than 16 bits per pixel must use a color set.) The bitmap may also include a reference to a color profile object that defines the color response characteristics of the device.

View-Related Objects

The end result of a drawing operation is the assignment of pixel values to the bitmap of a view device, followed by the transfer of those pixels to the screen or onto paper. In screen drawing or in printing, you can use transfer modes that either ignore or take into account the current pixel values of the bitmap, which themselves may be the products of previous drawing actions.

When you retrieve the bitmap property of a view device object, QuickDraw GX returns it to you as a bitmap shape that includes the information from the bitmap structure in the view device.

Bitmap shapes are described in the bitmap shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*. Color set objects and color profile objects are described in the chapter “Color and Color-Related Objects” in this book.

View Device Attributes

Each view device object has a set of attributes, a group of flags that influence device behavior. View device attributes allow you to make a device active or inactive, and to specify whether or not the pixel image needs to be stored in directly accessible memory. Table 7-3 lists the constants for the view device attributes and describes what each one means. The constants are defined in the `gxDeviceAttributes` enumeration.

Table 7-3 View device attributes

Constant	Value	Explanation
<code>gxDirectDevice</code>	<code>0x01</code>	If set, QuickDraw GX puts the pixel image of the view device's bitmap in directly accessible memory, if possible.
<code>gxRemoteDevice</code>	<code>0x02</code>	If set, QuickDraw GX puts the pixel image of the view device's bitmap in remote memory, such as on an accelerator card or video controller card, if possible.
<code>gxInactiveDevice</code>	<code>0x04</code>	If set, QuickDraw GX makes the device inactive, meaning that no drawing occurs on it. As an object, however, the device retains its existence and all its properties. QuickDraw GX sets this attribute for view device objects whose <code>GDevice</code> records mark them as inactive Macintosh graphics devices. The relation of graphics devices to view devices is described in the Macintosh environment chapter of <i>Inside Macintosh: QuickDraw GX Environment and Utilities</i> ; the <code>GDevice</code> record itself is described in <i>Inside Macintosh: Imaging with QuickDraw</i> .

The Default View Device Object

When you first create a view device object, you must assign it to a specific view group, and you must give it a reference to a bitmap shape—which you have set up—that describes the size, pixel depth, and location of the device's imaging area. Otherwise, the view device has these default properties:

- n A clip shape that is a full shape. The clip has no effect.
- n A mapping that is the identity mapping. This means that the upper left corner of the imaging area is at (0.0, 0.0) in global space, and the device resolution is 72 pixels per inch.
- n No attributes set.
- n An empty tag list.

For view devices that you do not create, QuickDraw GX defines their properties to be consistent with the physical devices they represent. See “View Device Objects and Physical Devices” (next).

View Device Objects and Physical Devices

View device objects are different from some other QuickDraw GX objects in that QuickDraw GX creates all that you need for ordinary drawing. Unless you are drawing offscreen or want unusual onscreen effects, you do not have to create a view device.

At startup, QuickDraw GX creates a view device object for each physical display device attached to the system, assigning the device to the onscreen view group (see “Onscreen and Offscreen View Groups” on page 7-29). QuickDraw GX sets the view device mapping and clip properties to reflect the device's pixel size, dimensions, and position in relation to other view devices. QuickDraw GX initializes the view device bitmap, assigning it a pixel image of appropriate size and pixel depth, and a color set and color profile based on information provided by the device's driver. (If the user changes the relative positions of display devices by some means such as the Monitors control panel, QuickDraw GX automatically updates the mappings and clips of the view devices to reflect the change.)

If you need information about any display device, you can first obtain a list of all the view device objects in the onscreen view group. You can then use the object references in that list to examine the properties of each device as needed.

View device objects are also associated with printers. To access the view device of a printer—if, for example, you want to mimic its characteristics with an offscreen view device—you use functions described in the advanced printing features chapter of *Inside Macintosh: QuickDraw GX Printing*.

For offscreen drawing, you do need to create your own view device objects and initialize all of their properties, including their bitmaps. Note that if you create a bitmap for offscreen drawing whose characteristics exactly match those of an onscreen view device, you can quickly transfer the results of your offscreen drawing to the screen by simply drawing that bitmap onscreen.

About View Group Objects

A **view group** object exists to relate view ports and view devices. It defines the set of view ports and view devices that can interact with each other, and it provides the basis for their relative positioning.

A view group represents a two-dimensional coordinate plane called *global space*. Global space imposes physical dimensions on and defines the spatial relationships among the view ports and view devices that belong to a view group.

You can have several view groups. Each defines a drawing world, allowing you to separate groups of view ports and view devices and draw to each group without conflict.

View Groups Have No Properties

As QuickDraw GX objects, view groups have no directly accessible properties. A view group is more like an ID number than an object containing information. Although you can at any time obtain a list of view ports and view devices that belong to a given view group, you can think of that information as coming from the individual view port and view device objects in the group, rather than from the view group object itself.

Likewise, the dimensional and positioning information imposed by a view group is all contained in the mapping matrices of the individual view ports and view devices that belong to the view group.

The only kinds of manipulation that you can perform directly on view group objects are creating them, disposing of them, and passing their references as function parameters.

Onscreen and Offscreen View Groups

QuickDraw GX creates one view group for you: the **onscreen view group**, whose reference is defined by the `gxScreenViewDevices` constant. It includes all physical screen devices. You draw to view ports in this view group for all onscreen drawing. If a transform object's view port list property is not changed from its default value, the only view port in the list is the default view port, which is in this view group. Thus, by default, shapes are drawn onscreen to view ports and onto view devices in this view group.

You can also create any number of offscreen view groups. For example, you can build an image by creating an offscreen view group that mirrors the onscreen view group and draw into view ports and onto view devices exactly as the drawing is done with the onscreen view group. The only difference is that your shapes appear in the bitmap of your offscreen view devices instead of onscreen. When you are ready to transfer those drawn shapes to the screen, you can draw the bitmaps of the offscreen view devices as bitmap shapes into view ports in the onscreen view group.

Clipping and Offscreen Drawing

For onscreen view ports attached to windows, QuickDraw GX takes care of restricting drawing to each window's visible areas, even in cases where windows overlap. When you create an offscreen view group with offscreen view ports, you need to take care of all clipping yourself, including cases in which view ports in different hierarchies overlap and those in front must clip those behind. \cup

To help you track all view ports and view devices for all onscreen and offscreen view groups, QuickDraw GX provides another predefined view group reference, defined by the `gxAllViewDevices` constant. You use it to specify all view groups when you want a list of all view ports or all view devices in all view groups. You cannot use this constant to set a view port or view device because `gxAllViewDevices` does not actually refer to a specific view group.

About Drawing, Coordinate Conversion, and Clipping

When you draw a QuickDraw GX shape, you are converting its internal representation into an image on an output device. QuickDraw GX uses information in the shape object and several other objects, including the view-related objects, to control how the shape is rendered. In brief, when you execute a drawing command QuickDraw GX follows this sequence of tasks:

1. It extracts the geometry of the shape object.
2. It applies stylistic and color information from the style object and ink object.
3. It applies the clip, and then the mapping, from the transform object.
4. It applies the mapping, and then the clip, from one or more view port objects.
5. It applies the mapping, and then the clip, from one or more view device objects.

The mapping operations specified in the transform, view port, and view device objects are concatenated during drawing, meaning that the operation is applied at one stage and the result is then used as input to the next calculation, and so on. Mapping is thus cumulative.

The clipping mechanism computes the intersection of the clip shapes of the transform object, view port objects, and view device objects. Each time a clip is applied, the visibility of a shape can only be further restricted.

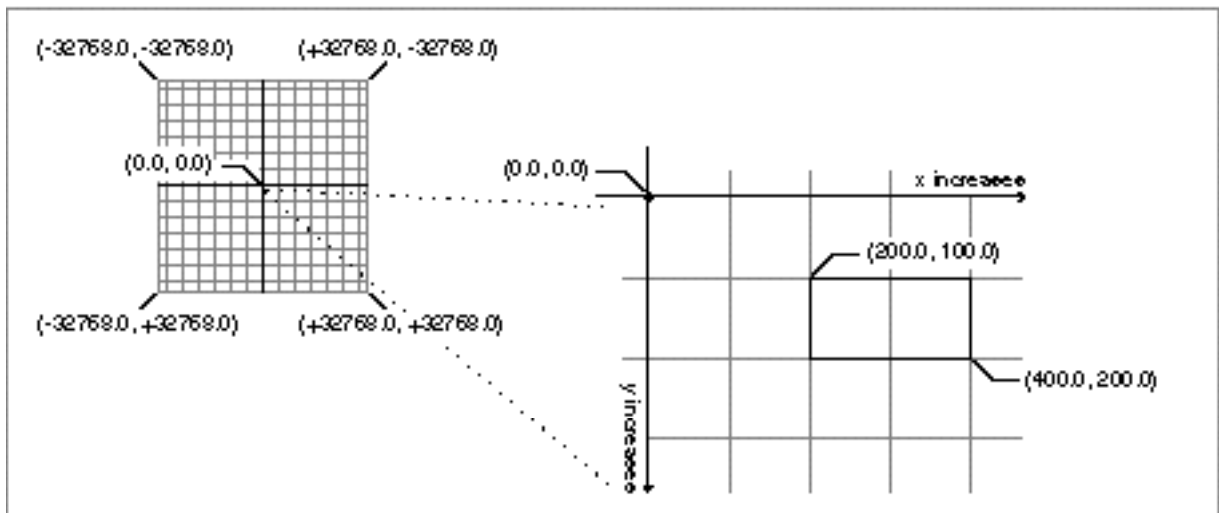
The following sections discuss this process in more detail, describing how QuickDraw GX uses four separate coordinate spaces to specify a shape during the stages of the drawing process, and what effects you can control at each stage.

QuickDraw GX Coordinates

All coordinates in QuickDraw GX are specified with fixed-point numbers in the range of $-32,768.0$ to approximately $32,768.0$. Fixed-point numbers and the functions for manipulating them are described in the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*. For any coordinate space, point $(0.0, 0.0)$ represents the origin of the space. Points that lie to the right of the origin increase in a positive direction along the x-axis; points that lie below the origin increase in a positive direction along the y-axis. Coordinates are always written in the order (x, y) .

Figure 7-13 shows the general layout of the QuickDraw GX coordinate plane, with an expanded portion that shows a rectangle 200 units wide by 100 units high, whose upper-left corner is at the point $(200.0, 100.0)$.

Figure 7-13 The QuickDraw GX coordinate plane



QuickDraw GX allows you to work in four coordinate spaces: geometry space, local space, global space, and device space. You can work separately in each space as appropriate for specific purposes; QuickDraw GX automatically converts among them when drawing.

The following discussion of coordinate spaces follows the progress of a drawing operation. It uses as an example the rendering of a single shape in a single window on a single view device. The shape, as finally displayed, is shown in Figure 7-19 on page 7-39. More complex possibilities, such as displaying in multiple windows and on multiple devices, are discussed as they arise.

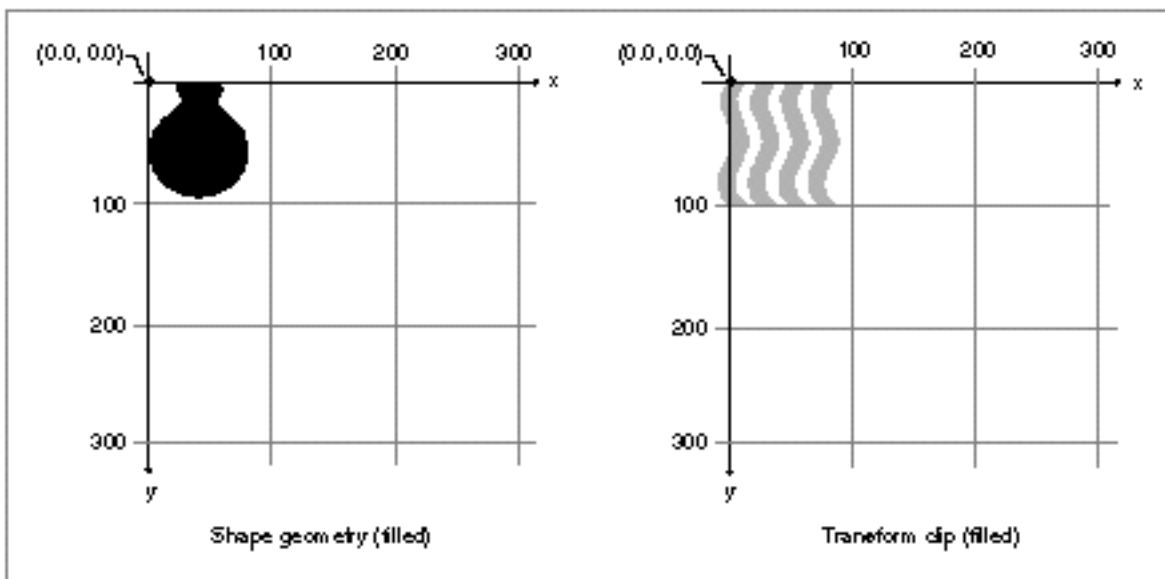
Geometry Space

Geometry space is the space within which the position and dimensions of a shape object are first defined. QuickDraw GX starts the drawing process by using the values in a shape's geometry; those values define the shape's fundamental dimensions, and their coordinate space is called geometry space.

No distance metric, such as points per inch, is defined for geometry space. Thus, the absolute size of a shape is undefined in geometry space. Also, every shape has its own geometry space; you cannot compare the sizes of two shapes based on their dimensions in geometry space alone. A rectangle 10 units wide in geometry space could end up ten times wider than a rectangle 100 units wide, once both have been drawn.

The left side of Figure 7-14 shows the geometry of a shape object, a filled path shape in the form of a vase. In geometry space, the vase is approximately 100 units by 100 units, and the upper-left corner of its bounding rectangle is at about (0.0, 0.0).

Figure 7-14 A shape geometry and a transform clip geometry



The right side of Figure 7-14 shows the geometry of the clip shape for a transform object. The clip shape is a filled path shape, of approximately the same dimensions and location as the vase shape. The next section shows how the clip shape modifies the appearance of the vase shape (assuming the vase shape object references the transform object containing this clip).

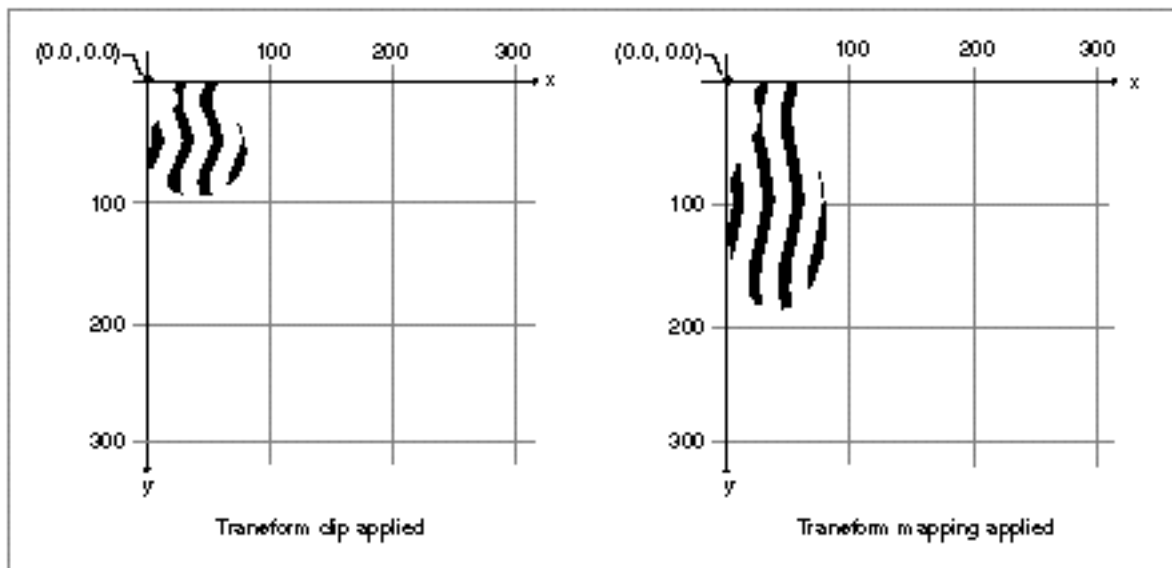
Local Space

Local space defines the location and dimensions of a shape after it has been modified by the mapping property of its associated transform object. Because mappings can translate, scale, rotate, skew, and otherwise distort geometries, the dimensions of a shape in local space can be quite different from what they are in geometry space.

As the first stage of drawing a shape, QuickDraw GX modifies the shape's geometry by applying information from the style object attached to the shape, and then applying first the clip and then the mapping contained in the transform object attached to the shape. Applying the mapping converts the shape from geometry space to local space. Because the transform clip is applied *before* the transform mapping, the dimensions of the clip shape are considered to be in geometry space. When you define the clip of a transform object, you size it and position it in terms of the dimensions of the shape's geometry.

The left side of Figure 7-15 shows the same vase shape as in Figure 7-14, this time after the transform clip has been applied to it. At this point the shape is still in geometry space—its overall position and dimensions unchanged, but its appearance modified by the clip.

Figure 7-15 Applying the transform's clip and mapping to a shape



The right side of Figure 7-15 shows the vase shape after the transform mapping has been applied to it. In this particular example, the only effect of the transform mapping is to scale the shape by a factor of 2.0 in the vertical direction, about an origin at (0.0, 0.0) in geometry space. The vase is now in local space.

Local space, like geometry space, has no metric; the absolute size of a shape object is still undefined after the transform mapping has been applied. You can, however, compare the sizes of two shape objects that share the same transform object. For example, if two path shapes have the same geometry and reference the same transform object, they are the same size.

You typically use the transform's clip and mapping for application-specific purposes related to moving, masking, and distorting shapes within a document. With the transform clip you define what parts of the shape geometry are to be visible, and with the transform mapping you choose how to move, orient, and distort that visible part of the shape, usually in relation to other shapes in the same document.

Several shape objects can reference the same transform object. This allows you to move, scale, rotate, and otherwise change an entire group of shapes in unison, by altering a single transform mapping.

Some shape types have specific additional definitions of local space:

- n Picture shapes, which consist of a hierarchy of other shapes, can have more than one transform object. In such a case, QuickDraw GX performs clipping and mapping operations on all transforms in turn from the bottom of the hierarchy to the top; the result of all those mapping transformations is considered local space for the shape. See the picture shapes chapter of *Inside Macintosh: QuickDraw GX Graphics* for information about the transform hierarchy in picture shapes.
- n Glyph shapes can have mappings in their geometries (tangent array) and in their associated style objects (text faces). In such cases, QuickDraw GX applies those mappings before applying the clip and mapping of the transform object to convert the glyph shape to local space. See the glyph shapes and typographic styles chapters of *Inside Macintosh: QuickDraw GX Typography* for information about the tangent array and text face mappings.

The transform object includes a reference to at least one view port object, and local space orients a shape within its view port. Local space is the coordinate system local to that view port—hence its name. Thus, the vase example in this section would have the same local coordinates—its bounding rectangle would have corners at about (0.0, 0.0) and (100.0, 200.0)—no matter how the view port itself might be scaled or distorted by its own mapping when it is converted to global space.

The fact that local space is the interior coordinate space of a view port means that you can compare the sizes of two shapes in local space even if they do not share the same transform—as long as they share the same view port. If two shapes have the same dimensions in local space and their transforms reference the same view port, they are the same size regardless of the actual values in their geometries or transform mappings.

Global Space

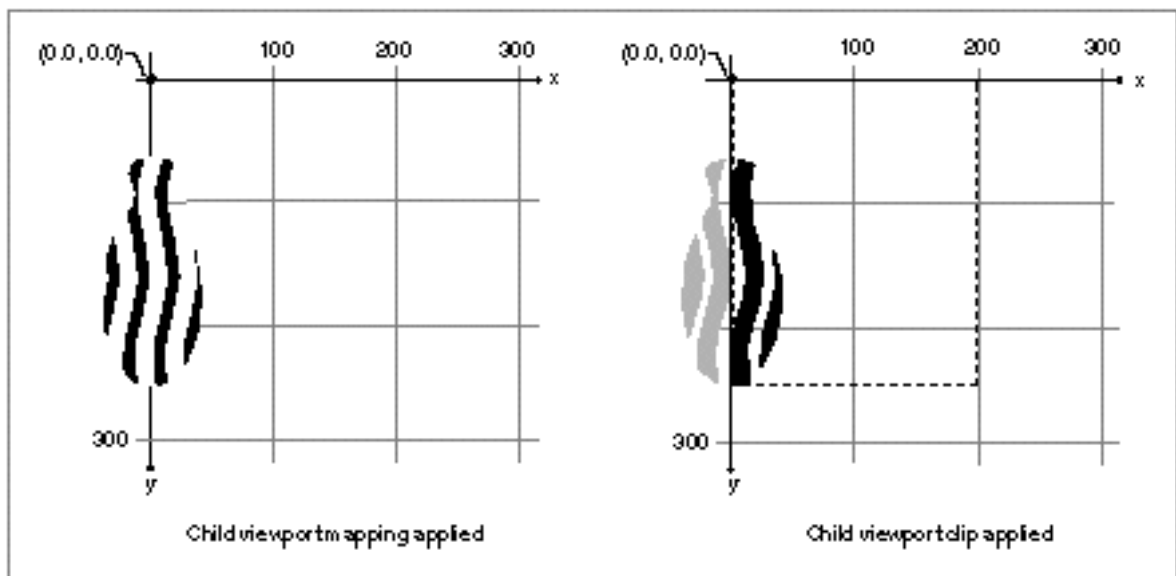
Global space contains the location and dimensions of a shape after the mapping in its associated view port has been applied. Global space defines the real-world location and dimensions of a shape; coordinate values in global space represent distance in points (72 per inch) from the origin of the view group that the view port is part of.

As this stage in drawing a shape, QuickDraw GX converts the shape from local space to global space. It modifies the shape's dimensions by applying first the mapping and then the clip contained in the view port object attached to the shape's transform. Because the view port clip is applied *after* the view port mapping, the dimensions of the clip shape are considered to be in global space. When you define the clip of a view port object, you size it and position it in terms of global space (the view port's position compared to view devices), not local space (the shape's position in its view port).

If the view port to which drawing occurs is a child view port in a view port hierarchy, QuickDraw GX performs mapping and clipping operations on all view ports in turn from that child view port through the top (root) view port; the result of all those mapping transformations is considered global space for the shape. See the section "Parent and Child View Ports" beginning on page 7-18 for information about view port hierarchies.

The example vase shape shown in the previous figures is drawn into the child view port of a simple two-level hierarchy. The left side of Figure 7-16 shows the vase shape after the child view port mapping has been applied to it. In this particular example, the effect of the view port mapping is to move the shape downward and to the left by approximately 50 units, representing a scrolling of the shape from its original position. There is no scale factor or other distortion in this case, so the dimensions of the shape are unchanged. The shape is not yet in global space, however, because another mapping (from the parent view port) must be applied.

Figure 7-16 Applying the child view port's mapping and clip to a shape

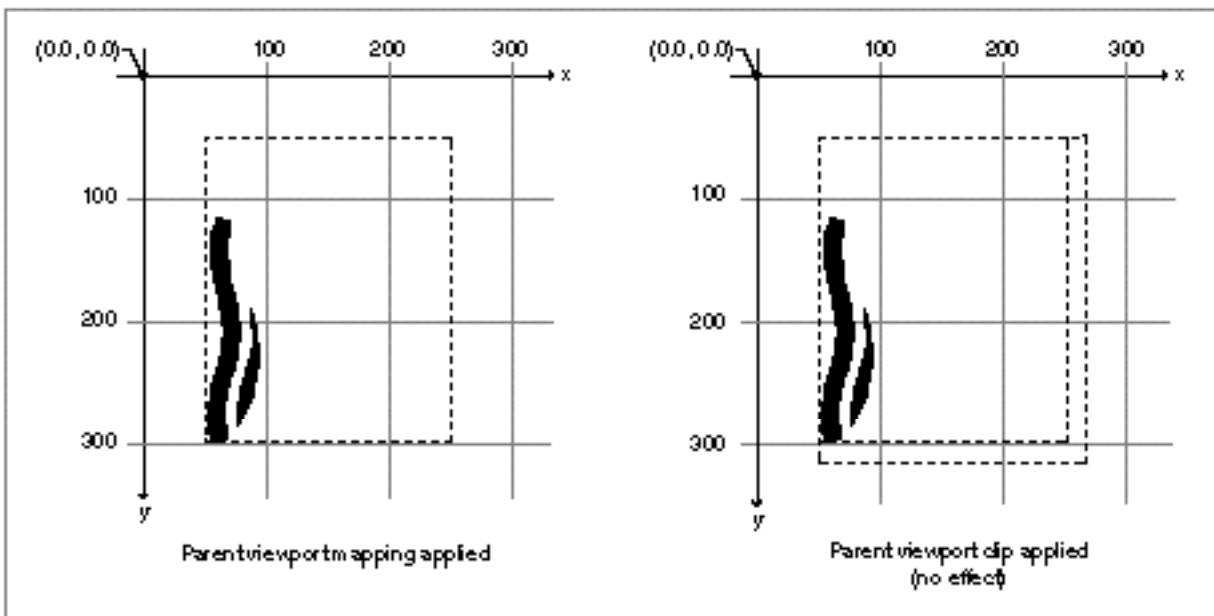


The right side of Figure 7-16 shows the vase shape after the child view port clip has been applied to it. The view port clip in this case is a rectangle that defines the visible portion of the child view port. As Figure 7-16 shows, the clip cuts out the left half of the vase (shaded gray), meaning that part of the shape has been scrolled out of view in its view port. The clip's dimensions, although not yet in global space (because the parent view port mapping has not yet been applied), are "global" to the child view port; changing the child view port's mapping, for example, does not change the position of its clip in relation to its parent view port. Therefore, to scroll a shape in a view port, you need only change the view port's mapping, not its clip.

This example shows a single shape drawn into a single view port, but more complex arrangements are possible. For example, as shown in Figure 7-3 on page 7-10, a single transform object can reference several view port objects, allowing a single shape to appear simultaneously (perhaps with different scaling or orientation) in several view ports.

Figure 7-17 completes the process of conversion from local to global space. The left side of Figure 7-17 shows the vase shape after the parent view port mapping has been applied to it. In this example, the effect of the view port mapping is to move the shape to the right and downward by approximately 50 units, representing the actual location of the shape in global space. Again, there is no scale factor or other distortion applied in this case, so the dimensions of the shape are unchanged. Because this view port is the root view port, the shape is now in global space and its dimensions can be measured. The visible part of the shape is approximately 50 points by 200 points in size, or about 0.7 by 2.8 inches.

Figure 7-17 Applying the parent view port's mapping and clip to a shape



The right side of Figure 7-17 shows the vase shape after the parent view port clip has been applied to it. This view port clip is a rectangle that defines, in global space, the content area of the window to which the parent view port is attached. As is typical for a simple window that supports scrolling, the clips of the child view port and parent view port differ only by the areas of the scroll bars; the child view port clip fits inside the scroll bars so that drawing into it does not obliterate the scroll bars. In this case, the application of the parent view port clip has no effect on the visibility of the vase shape because the child view port clip is entirely contained within it.

As this example shows, you typically use the parent view port's mapping to position the window you are drawing into, and its clip to restrict drawing to the interior of the window. You use mappings of child view ports to scroll, scale, or move shapes in relation to the parent view port, and you use their clips to restrict the shapes' visibilities in relation to the parent view port. If a parent view port is attached to a window (through the `GXNewWindowViewPort` call), QuickDraw GX itself manipulates both the clip and mapping of the parent view port to make sure its location and drawable area correspond to the visible parts of the content area of the window. (Strictly speaking, QuickDraw GX prevents drawing from occurring outside of the visible part of the content area of the window, but it does not necessarily use the view port's clip to do so; if you retrieve the clip of a window view port, it is not guaranteed to be equal to either the window's port rectangle or its visible region.)

Global space is view-group space. Keep in mind these ways in which view groups and global space define the interactions among view ports and view devices:

- n Once a shape's dimensions have been converted to global space, it has an absolute size and a specific spatial relationship to all other shapes in that view group, whether or not the shapes share the same local space (view port).
- n Global-space dimensions are device-independent and therefore resolution independent; for typical drawing operations, you need never know the resolutions of the devices you are drawing to.
- n Within a view group, the clips of view ports and view devices can overlap in any combination. Drawing occurs automatically wherever the visible portions of any view port and any view device in that view group overlap.
- n More than one view group can exist simultaneously, allowing for offscreen drawing. Furthermore, the view ports referenced by the transform of a single shape need not all be in the same view group, allowing for simultaneous onscreen and offscreen drawing of a shape.

To draw the device-independent shapes in a view group with maximum accuracy on view devices of varying positions and resolutions requires conversion from global space to device space, as described next.

Device Space

Device space defines the location and dimensions of a shape as displayed on a particular output device. The upper-left corner of the displayable area of a view device is at coordinate (0.0, 0.0) in device space. Unit distance between coordinates in device space represents one picture element, or pixel.

The view device's mapping defines both its location in global space (as a translation factor) and its pixel size (as a scaling factor). For example, if your device is a 144 pixels-per-inch high-resolution monitor, QuickDraw GX converts global space to device space when drawing by scaling each global-space point by 2.0, which is $144/72$. By default, if there is a single view device in a view group, the translation value in its mapping is 0, meaning that point (0.0, 0.0) in device space is also point (0.0, 0.0) in global space. The view device's clip is a (usually rectangular) shape representing the displayable area of the device.

As the final stage in drawing a shape, QuickDraw GX converts the shape from global space to device space. It modifies the shape's dimensions by applying first the mapping and then the clip of any view device object in the same view group whose clip overlaps the view port clip.

The example vase shape shown in the previous figures is drawn onto a single view device. The left side of Figure 7-18 shows the vase shape after the view device mapping has been applied to it. In this example, the view device mapping specifies no translation, but the pixel resolution is 144 ppi so it scales the shape by 2.0. The shape is now in device space, and its visible part is approximately 100 by 400 pixels in size (which is still about 0.7 by 2.8 inches).

Figure 7-18 Applying the view device's mapping and clip to a shape

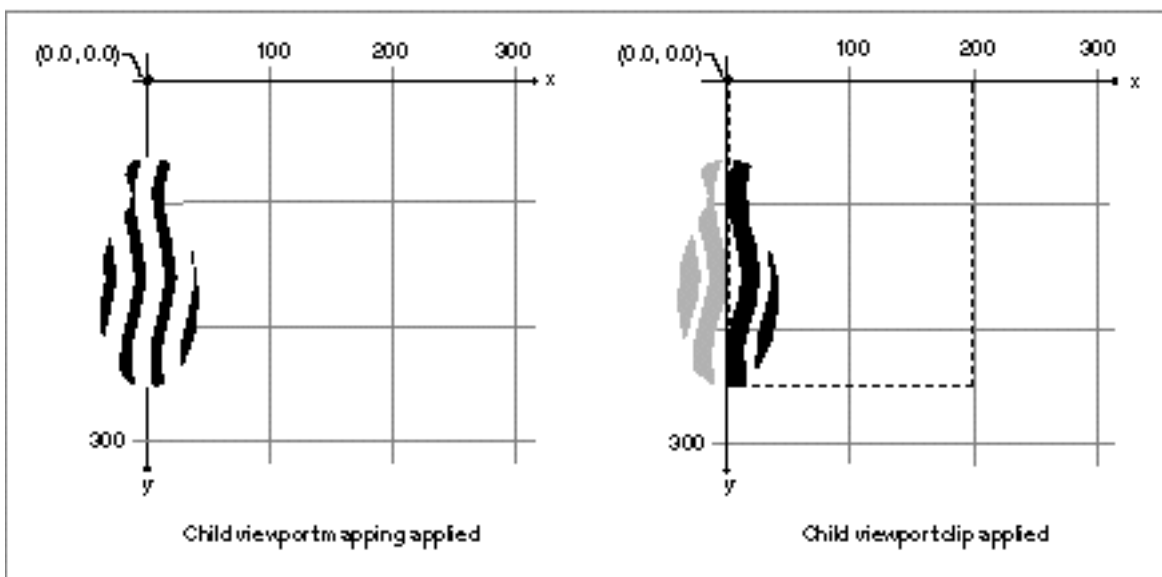
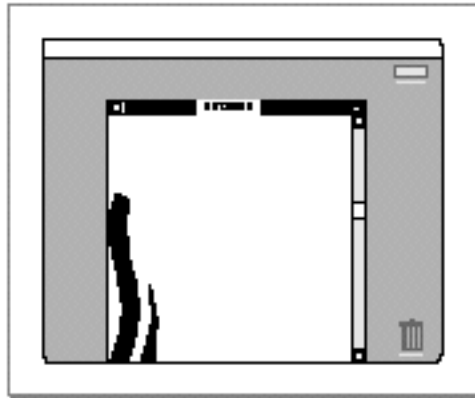


FIGURE 7-18

The right side of Figure 7-18 shows the vase shape after the view device clip has been applied to it. The view device clip for the monitor represents its imaging area, exclusive of the menu bar. In this case, the view device clip cuts off part of the visible area of the view port, so the lower part of the vase shape (shaded gray in Figure 7-18) is not drawn on this device.

Figure 7-19 shows the example vase shape as actually displayed, after all clipping and coordinate conversions have been applied. The clipped and stretched vase shape is partially scrolled out of view in its window, and the lower part of the window is clipped by the bottom edge of the monitor.

Figure 7-19 The shape as finally displayed



It is seldom necessary to work in device space because QuickDraw GX performs this conversion for you. QuickDraw GX also handles modifications to the view device mappings to match the monitor configuration. For example, if you use the Monitors control panel to change the relative positions of monitors on a Macintosh system, QuickDraw GX handles the changes for you.

Using View-Related Objects

View-related objects define the drawing environment for a QuickDraw GX application. Often, you set up your view ports, view devices and view groups when you set up other application structures. Because of the interrelationships between these objects, setting up an offscreen or onscreen environment and manipulating it involves creating and setting up several objects.

This section describes how you can

- n create and use view ports, and analyze shapes in view ports
- n create and use view devices, and analyze shapes on view devices
- n create and use view groups for offscreen drawing, and analyze shapes in view groups

Using View Ports

This section demonstrates how to use QuickDraw GX view ports. It shows how you can

- n create and manipulate view port objects and their properties
- n get and set a view port's clip and mapping
- n set up a view port hierarchy attached to a window
- n support scrolling in a window
- n identify the view devices of a view port and the view ports of a shape
- n measure a shape in the local space of a view port

Creating and Manipulating View Port Objects

QuickDraw GX provides several functions with which you can create a new view port. To create a view port that is the root view port of a hierarchy and is attached to a Macintosh window, you use the function `GXNewWindowViewPort`, described in the Macintosh environment chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*. To create child view ports for that root parent, or to create a root view port for offscreen drawing, you use the `GXNewViewPort` function. You can also create a new view port object by copying an existing one with the `GXCopyToViewPort` function; see Listing 7-2 on page 7-44 for an example of this method.

Replacing the default view port

If your application draws only within windows, you may want to replace the default transform object (which references the default view port) for each shape type. You could replace it with a transform object that directly references a window view port (a view port attached to a Macintosh window). Alternatively, you could replace it with a transform that references a view port you have designated as the “current” view port; you could then redirect drawing to a window view port by assigning the window view port as the parent of the current view port. Or you could take a different approach and explicitly assign a child view port of a window view port to each shape as it is created, using the `GXSetShapeViewPorts` function. ^u

Once you have created a view port object, you can customize its features using the techniques described in the following section.

You can test if two view port-object references refer to the same view port object by simply testing the references for equality. You can also test a view port for equality with another view port with the `GXEqualViewPort` function. For two view port objects to be equal, their mappings, clips, dithers, halftones, attributes, parent view ports, and view groups must be identical; if one view port is attached to a window, the other view port must be attached to the same window. The tag lists or child view ports of the view ports need not be identical. View port object copies created with the `GXCopyToViewPort` function are always equal to the view port from which they were copied.

To delete your application's reference to a view port object, call the `GXDisposeViewPort` function. Because view port objects have no owner count, calling `GXDisposeViewPort` actually releases the memory allocated for that view port object, and invalidates all other references to it. Therefore, once you have disposed of a view port, other transform objects that reference that view port will have invalid view port references in their view port lists. This causes no error when you try to draw; drawing simply does not occur to view ports whose references are invalid.

The following code fragment first creates a Macintosh window (`sampleWindow`), and then uses `GXNewWindowViewPort` to create a view port attached to it. When it no longer needs them, the code disposes of the view port and then the window. The `NewWindow` function and its parameters, and the `DisposeWindow` function, are described in the Window Manager chapter of *Inside Macintosh: Macintosh Toolbox Essentials*.

```
sampleWindow = NewWindow( nil, &windowRect, "\p", true,
                          documentProc, (WindowPtr)-1L, true, 0L );
aViewPort = GXNewWindowViewPort(sampleWindow);
.
. /* use the window and view port */
.
GXDisposeViewPort(aViewPort);
DisposeWindow(sampleWindow);
```

The following line of code creates a view port that is not attached to a window. You might use this call to create a view port that is to be the child of another view port. The code assigns the new view port to the `gxScreenViewDevices` view group, the view group for all onscreen drawing:

```
myChildViewPort = GXNewViewPort(gxScreenViewDevices);
```

A more general way to assign the view group parameter is to first call the `GXGetViewPortViewGroup` function to determine the view group of the intended parent view port, and use that result as the parameter for `GXNewViewPort`. See, for example, Listing 7-5 on page 7-47.

The `GXNewViewPort` function is described on page 7-70. The `GXCopyToViewPort` function is described on page 7-72. The `GXEqualViewPort` function is described on page 7-73. The `GXDisposeViewPort` function is described on page 7-71.

Manipulating View Port Object Properties

This section describes how to manipulate the dither, halftone, view group, attributes, and tag list properties of a view port object:

- n To manipulate the dither level, you use the functions `GXGetViewPortDither` and `GXSetViewPortDither`.
- n To manipulate the halftone structure, you use the functions `GXGetViewPortHalftone` and `GXSetViewPortHalftone`.
- n To manipulate the view group reference, you use the functions `GXGetViewPortViewGroup` and `GXSetViewPortViewGroup`.
- n To manipulate the view port attributes, you use the functions `GXGetViewPortAttributes` and `GXSetViewPortAttributes`.
- n To manipulate the view port tag list, you use the functions `GXGetViewPortTags` and `GXSetViewPortTags`.

How to manipulate other view port properties is described in subsequent sections, starting with “Getting and Setting a View Port’s Clip and Mapping” on page 7-44.

Getting and Setting a View Port’s Dither, Halftone, and Attributes

Listing 7-1 is an example of code that sets the dither level, halftone structure, and view port attributes of the view port `myViewPort`. For the halftone structure (`myHalfTone`), the code sets all of its values, including the background color and the dot color in HSV color space. The tint type selected, however, is luminance tint, meaning that only the lightness of the input color is used to calculate the proportion of dot and background to use for the halftone. The attributes specify a grayscale view port, meaning that the dot and background colors are also drawn in gray.

Listing 7-1 Changing a view port’s dither, halftone, and attributes

```
gxViewPort myViewPort
gxHalftone myHalfTone;

GXSetViewPortAttributes(myViewPort, gxGrayPort);
GXSetViewPortDither (myViewPort, 4);

myHalfTone.angle = ff(6);
myHalfTone.frequency = ff(24);
myHalfTone.method = gxDispersedDot;
myHalfTone.tinting = gxLuminanceTint;
myHalfTone.tintSpace = gxHSVSpace;
```


View-Related Objects

```

myHalfTone.backgroundColor.space = gxHSVSpace;
myHalfTone.backgroundColor.profile = nil;
myHalfTone.backgroundColor.element.hsv.value = 0xFFFF;
myHalfTone.backgroundColor.element.hsv.saturation = 0xCCCD;
myHalfTone.backgroundColor.element.hsv.hue = 0x8000;

myHalfTone.dotColor.space = gxHSVSpace;
myHalfTone.dotColor.profile = nil;
myHalfTone.dotColor.element.hsv.value = 0xFFFF;
myHalfTone.dotColor.element.hsv.saturation = 0xAD1C;
myHalfTone.dotColor.element.hsv.hue = 0xE4F9;

GXSetViewPortHalftone(myViewPort, &myHalfTone);

```

Note

Dithers and halftones are mutually exclusive. The halftone in this example overrides the dither, so dithering is not performed at drawing. u

The `GXGetViewPortDither` function is described on page 7-80; the `GXSetViewPortDither` function is described on page 7-80.

The `GXGetViewPortHalftone` function is described on page 7-81; the `GXSetViewPortHalftone` function is described on page 7-82.

The `GXGetViewPortAttributes` function is described on page 7-89; the `GXSetViewPortAttributes` function is described on page 7-90.

Getting and Setting a View Port's View Group

Listing 7-2 demonstrates the use of the `GXSetViewPortViewGroup` function. It is part of a routine that creates an offscreen view group (`newGroup`) that is a copy of an existing view group (`group`). For each view port in the onscreen view group, the routine creates a copy. It then uses `GXSetViewPortViewGroup` to assign the proper view group to the new view port. (Listing 7-10 on page 7-54 shows another part of the same routine.)

The routine uses the `count` variable to decrement through the list of view ports (`oldList`, retrieved through two consecutive calls to `GXGetViewGroupViewPorts`) belonging to the onscreen view group. The code simultaneously builds, for its own purposes, a list (`newList`) of view ports for the offscreen view group, using `GXCopyToViewPort` and `GXSetViewPortViewGroup` to copy each view port into the offscreen view group and set its view group property.

Listing 7-2 Copying the view ports from one view group to another

```

long          portCount = GXGetViewGroupViewPorts(group, nil);
long          count = portCount;
gxViewPort   *oldPortList = (void *)NewPtr(portCount *
                                         sizeof(gxViewPort));

gxViewPort   *oldList = oldPortList;
gxViewPort   *newPortList = (void *)NewPtr(portCount *
                                         sizeof(gxViewPort));
gxViewPort   *newList = newPortList;
GXGetViewGroupViewPorts(group, oldPortList);
while (count-- > 0)
    GXSetViewPortViewGroup(*newList++ = GXCopyToViewPort(nil,
                                                           *oldList++), newGroup);

```

The `GXGetViewPortViewGroup` function is described on page 7-88. The `GXSetViewPortViewGroup` function is described on page 7-88.

Getting and Setting a View Port's Tag References

You can examine the list of references to tag objects currently associated with a view port object using the `GXGetViewPortTags` function. Once you create a tag object, you can attach it to a view port object using the `GXSetViewPortTags` function. You can attach as many tag objects as you like to a view port object.

Tag objects and the basic functions for manipulating them are described in the chapter “Tag Objects” in this book. That chapter also lists the common tag types defined and reserved by Apple Computer, Inc.

The `GXGetViewPortTags` function is described on page 7-91. The `GXSetViewPortTags` function is described on page 7-92.

Getting and Setting a View Port's Clip and Mapping

The clip and mapping properties of a view port control the visibility and location of its contents. For onscreen view ports attached to Macintosh windows, you do not directly set the clip or mapping properties; you move or resize the window with Window Manager calls, and QuickDraw GX automatically updates the view port's clip and mapping. For child view ports of window view ports, however, and for all offscreen view ports, you must set the clip and mapping yourself.

You use the functions `GXGetViewPortMapping`, and `GXSetViewPortMapping` to set a view port mapping to move the contents of the view port, such as when scrolling. You also set the view port mapping to provide scaled, rotated, or otherwise altered views of the view port's contents. Listing 7-3 shows an example that uses those functions plus `GXGetViewPortClip` and `GXScaleMapping` to scale the view port `myViewPort` to 200 percent of its original size, about an origin at the center of the view port's clip.

Listing 7-3 Changing a view port's mapping

```

gxViewPort      myViewPort
gxMapping       myViewPortMapping;
gxShape         myViewPortFrame;
gxPoint         center;

GXGetViewPortMapping(myViewPort, &myViewPortMapping);
myViewPortFrame = GXGetViewPortClip(myViewPort);
GXGetShapeCenter(myViewPortFrame, 0L, &center);
GXScaleMapping(&myViewPortMapping, ff(2), ff(2),
               center.x, center.y);
GXSetViewPortMapping(myViewPort, &myViewPortMapping);
GXDisposeShape(myViewPortFrame);

```

Note that, because the `GXGetViewPortClip` function creates a shape object, the code in Listing 7-3 disposes of the shape after using it. The `GXGetShapeBounds` function is described in the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*; the `GXSetShapeMapping` function is described in the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

Getting the global mapping

If a view port is a child view port in a hierarchy, its mapping converts from local space into the local space of its parent view port, not directly into global space. If you want to determine the resultant mapping obtained by concatenating the mappings of a view port and all its parents—a mapping from local space all the way into global space—you can use `GXGetViewPortGlobalMapping`, which is described on page 7-79. For an example of its use, see Listing 7-11 on page 7-57. u

You can use the `GXSetViewPortClip` function to set a view port clip to initialize or change the visible area of the view port. Listing 7-4 is a routine that sets up the clip of a child view port (`gcontentViewPort`) whose parent is the root view port attached to a Macintosh window (`theWindow`). The routine makes the clip the same size as the window's content area, minus the width of the scroll bars on the window's side and bottom.

The listing uses the application-defined function `GetWindowBoundsShape` to determine the rectangle shape corresponding to the content area of the window. That function retrieves a QuickDraw rectangle corresponding to the port rectangle of the window, and then converts it to a QuickDraw GX rectangle using the `GXConvertQDPoint` function.

Listing 7-4 Setting a view port clip

```

void ResetContentViewPortClip (WindowPtr theWindow)
{
    gxRectangle    viewRect;
    gxShape        contentViewPortClipShape;

    /* get the size of the window port rect */
    GetWindowBoundsShape(theWindow, &viewRect);

    /* Adjust the rectangle to accommodate the scroll bars */
    viewRect.right -= ff(kScrollBarWidth - 1);
    viewRect.bottom -= ff(kScrollBarWidth - 1);

    /* assign it as the clip shape */
    contentViewPortClipShape = GXNewRectangle(&viewRect);
    GXSetViewPortClip(gcontentViewPort, contentViewPortClipShape);
    GXDisposeShape (contentViewPortClipShape);
}

```

The `GXConvertQDPoint` function is described in the Macintosh environment chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*. The `GXNewRectangle` function, which creates a rectangle shape, is described in the geometric shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*.

The `GXGetViewPortClip` function is described on page 7-74; the `GXSetViewPortClip` function is described on page 7-75.

The `GXGetViewPortMapping` function is described on page 7-77; the `GXSetViewPortMapping` function is described on page 7-78.

Setting Up the View Port Hierarchy for a Window

Setting up a view port hierarchy means assigning the appropriate parent view port and child view port references to all view ports involved. The functions you use are `GXGetViewPortParent`, `GXSetViewPortParent`, `GXGetViewPortChildren`, and `GXSetViewPortChildren`. Take these steps to set up a simple hierarchy in which a child view port is used for drawing a window's content:

1. Create the child view port in the window view port's view group.
2. Create a clip shape and assign it to the child view port. Set the child view port's mapping.
3. Assign the window view port as the parent of the child view port.
4. Dispose of the clip shape.

Note that you do not have to add the child view port to the window view port's list of children; when you set the parent view port property of the child view port, QuickDraw GX adds the child view port to the parent's list of child view ports.

Listing 7-5 is an example that sets up such a hierarchy. It creates a view port with the `GXNewViewPort` function, and uses `GXGetViewPortViewGroup` to find out what view group to assign it to. The code assigns properties to the view port with `GXSetViewPortClip`, `GXSetViewPortMapping`, and `GXSetViewPortParent`. The window view port is `windowParentViewPort`, and the rectangle `viewRect` defines a clipping area that is the size of the window minus the area reserved for scroll bars.

Listing 7-5 Setting up a view port for a window

```

gxRectangle    viewRect;
gxViewPort    windowParentViewPort;
gxShape        contentViewPortShape;
. /*
.     Create window view port with GXNewWindowViewPort. Make
.     viewRect equal to port rectangle minus scroll bars.
. */
gcontentViewPort = GXNewViewPort
                    (GXGetViewPortViewGroup(windowParentViewPort));
contentViewPortShape = GXNewRectangle(&viewRect);
GXSetViewPortClip(gcontentViewPort, contentViewPortShape);
GXSetViewPortMapping(gcontentViewPort, nil);
GXSetViewPortParent(gcontentViewPort, windowParentViewPort);
GXDisposeShape (contentViewPortShape);

```

Once you have set up a hierarchy, if you want to draw into the child view port—and thus onscreen—you must place a reference to the child view port in a transform object's view port list, and the shapes you draw must reference that transform.

The `GXGetViewPortParent` function is described on page 7-84; the `GXSetViewPortParent` function is described on page 7-84.

The `GXGetViewPortChildren` function is described on page 7-86; the `GXSetViewPortChildren` function is described on page 7-87.

Supporting Scrolling in a Window

To support scrolling in a view port attached to a Macintosh window, you need to create a child view port of the window view port, and draw into it rather than into its parent. QuickDraw GX prevents you from changing the mapping or clip of a view port directly attached to a Macintosh window.

When the user scrolls the window, you manipulate the child view port's mapping to scroll the content. When the user resizes the window, you manipulate the child view port's clip to fit the new window shape. When the user moves the window, you do nothing; QuickDraw GX takes care of positioning both parent and child view ports.

Listing 7-6 is an example of a scrolling routine that scrolls the contents of a child view port (`gcontentViewPort`) by specified vertical and horizontal amounts (`hScroll` and `vScroll`), in response to mouse-down events in the scroll bars of a window (`theWindow`). The event-dispatching routine calls this scrolling routine after it has calculated how much scrolling is required. After the scrolling routine executes, a separate routine (not shown) updates the appearance of the scroll bars.

Listing 7-6 Supporting scrolling in a child view port

```
void DoScroll(WindowPtr theWindow, short hScroll, short vScroll)
{
    Rect        scrollRect;
    Point       scrollPt;
    RgnHandle   myRgn;
    gxMapping   viewPortMapping;

    if ((hScroll == 0) && (vScroll == 0)) return;

    /*
     * Get the child view port's mapping, adjust it for the
     * horizontal and vertical scroll, then reassign it to the
     * view port. The next drawing action will then reflect the
     * scrolled positions of shapes in the view port.
     */
    GXGetViewPortMapping(gcontentViewPort, &viewPortMapping);
    MoveMapping(&viewPortMapping, ff(hScroll), ff(vScroll));
    GXSetViewPortMapping(gcontentViewPort, &viewPortMapping);

    /*
     * Shift the pixels representing the already drawn contents of
     * window, so that less will have to be drawn when the window
     * is updated. Only the parts scrolled into view (specified by
     * the update region myRgn) will need to be redrawn.
     */
    scrollRect = theWindow->portRect;
    scrollRect.right -= (kScrollBarWidth-1);
    scrollRect.bottom -= (kScrollBarWidth-1);

    SetPort(theWindow);
    myRgn = NewRgn();
    ScrollRect(&scrollRect, hScroll, vScroll, myRgn);
}
```

View-Related Objects

```

    /* update origin position in app's extended window record */
    SetPt(&scrollPt, hScroll, vScroll);
    AddPt(scrollPt, &((MyWindowPeek)theWindow)->origin);

    /* redraw the window and dispose of the region handle */
    DrawWindow(theWindow);
    DisposeRgn(myRgn);
}

```

Identifying a View Port's View Devices

The `GXGetViewPortViewDevices` function returns a list of all view devices that could be affected by shapes drawn in a given view port. Within the view group of the view port, all view device objects whose clip areas overlap the clip area of the view port appear in the list returned by this function.

You can use `GXGetViewPortViewDevices` to determine whether a given view device can be affected by drawing into a given view port. You can also use it to examine the properties of all devices that you might draw to, perhaps in order to assign appropriate properties to offscreen view devices.

Listing 7-7 is a library function (`SetShapeFastXorTransfer`) that uses another library function (`SetInkFastXorTransfer`) to set up a shape's color and an XOR transfer mode so that drawing with that color will cause a specified highlight color to replace the background color in the destination. The `SetInkFastXorTransfer` function is not shown here. Listing 7-7 is shown because it uses `GXGetViewPortViewDevices` to get a list of the view devices a view port can draw to, although it actually uses only the first view device in the list.

Listing 7-7 Setting a shape color for XOR highlighting

```

void SetShapeFastXorTransfer(gxShape source, gxColor *background,
                           gxColor *result)
{
    long          viewPortCount, viewDeviceCount;
    void          *buffer;
    gxViewPort    vp;
    gxViewDevice  vd;
    gxInk         inky;

    /* get size of view port list, then allocate buffer for it */
    viewPortCount = GXGetTransformViewPorts(
                    GXGetShapeTransform(source), nil);
    buffer = NewPtr(sizeof(gxViewPort) * viewPortCount);
}

```

View-Related Objects

```

/* check for memory error (not shown), then get list itself */
GXGetTransformViewPorts(GXGetShapeTransform(source),
                        (gxViewport *)buffer);

/* get no. of view devices, then allocate buffer for list */
viewDeviceCount = GXGetViewportViewDevices(
                    vp = *(gxViewport *)buffer, nil);
.
. /* check for memory error (not shown), then get list itself */
.
GXGetViewportViewDevices(vp, (gxViewDevice *)buffer);
vd = *(gxViewDevice *)buffer;
DisposePtr(buffer);

/* get shape's ink; if shared, assign a copy of the ink */
if (GXGetInkOwners(inky = GXGetShapeInk(source)) > 1)
{
    GXSetShapeInk(source, inky = GXNewInk());
    GXDisposeInk(inky);
}
/* set ink's transfer mode and suppress dithering */
SetInkFastXorTransfer(inky, vd, vp, background, result);
GXSetShapeInkAttributes(source,
                        GXGetShapeInkAttributes(source) |
                        gxSuppressDitherInk);
}

```

The `GXGetViewportViewDevices` function is described on page 7-94.

Identifying a Shape's View Ports

The `GXGetShapeGlobalViewPorts` function returns a list of all view ports that a shape could actually appear in if it were drawn. If a shape's transform references a view port, and if that view port's clip does not totally exclude the shape from the visible part of the view port, the view port appears in the list returned by this function.

You can use `GXGetShapeGlobalViewPorts` to avoid the overhead of drawing shapes that cannot be visible. You can also use it as an input to the `GXGetShapeGlobalViewDevices` function to determine all the devices on which a given shape can appear.

The `GXGetShapeGlobalViewPorts` function is described on page 7-95. The `GXGetShapeGlobalViewDevices` function is described on page 7-115.

Measuring a Shape in Local Space

The `GXGetShapeLocalBounds` function measures the bounding rectangle of a shape in local coordinates—that is, after the transform mapping has been applied to the shape geometry. You can use `GXGetShapeLocalBounds` to compare the positions and sizes of two shapes in the same view port, even if they do not share the same transform object. (To compare the positions and sizes of two shapes in different view ports, use the `GXGetShapeGlobalBounds` function; to measure a shape on a view device, use `GXGetShapeDeviceBounds`.)

Listing 7-8 is a function in a shape-editing program. It draws a gray box representing the bounding rectangle for each shape in a list of shapes passed to it. It calls `GXGetShapeLocalBounds` for each shape, and then defines and draws a rectangle shape whose geometry matches that bounding rectangle. Regardless of how each shape has been modified by its own transform mapping, `GXGetShapeLocalBounds` returns a rectangle (whose default transform has an identity mapping) that exactly matches the transformed shape's bounding rectangle when drawn in the view port.

Listing 7-8 makes use of the library function `SetShapeCommonColor` to set the bounding rectangle's color.

Listing 7-8 Locating the bounding rectangles of a list of shapes in a view port

```
void ShowLocalBounds(gxShape *plstShape, long shapeCount)
{
    register gxShape    *pShape, rectShape;
    gxRectangle         bounds;

    pShape = plstShape + shapeCount - 1;    /* no. of last shape */

    /* define a framed gray rectangle shape for the bounds */
    rectShape = GXNewShape(gxRectangleType);
    GXSetShapeFill(rectShape, gxClosedFrameFill);
    SetShapeCommonColor(rectShape, gxGray);

    /* go through shape list, get and draw local bounds for each */
    while (shapeCount--)
    {
        GXGetShapeLocalBounds(*pShape--, &bounds);
        GXSetRectangle(rectShape, &bounds);
        GXDrawShape(rectShape);
    }
    GXDisposeShape(rectShape);
}
```

The `GXGetShapeLocalBounds` function is described on page 7-96.

The `GXGetShapeGlobalBounds` function is described on page 7-125; the `GXGetShapeDeviceBounds` function is described on page 7-116.

Using View Devices

This section demonstrates how to use QuickDraw GX view devices. It shows how you can

- n create and manipulate view device objects and their properties
- n get and set a view device's clip and mapping
- n identify the view devices of a shape
- n measure a shape in the device space of a view device
- n hit-test a shape on a device

Creating and Manipulating View Device Objects

Normally, your application needs to create view device objects only for offscreen drawing. QuickDraw GX creates view device objects for all attached screen devices at startup. If you do need to create a view device object, QuickDraw GX provides the `GXNewViewDevice` function, to which you must supply a view group reference and a bitmap shape representing the device's imaging area and characteristics. You can also create a new view device object by copying an existing one, using the `GXCopyToViewDevice` function.

Once you have created a view device object, you can customize its features using the techniques described in the following sections.

You can test if two view device-object references refer to the same view device object by simply testing the references for equality. You can also test a view device for equality with another view device with the `GXEqualViewDevice` function. For two view device objects to be equal, their clips, mappings, bitmap shapes, and attributes must be identical, and they must be in the same view group, represent the same Macintosh graphics device (same `GDevice` record), and point to the same pixel image, color set, and color profile. Their tag lists need not be identical. View device object copies created with the `GXCopyToViewDevice` function are always equal to the view device from which they were copied.

To delete your application's reference to a view device object, call the `GXDisposeViewDevice` function. Because view device objects have no owner count, calling `GXDisposeViewDevice` actually releases the memory allocated for that view device object, and invalidates all other references to it.

Listing 7-9 is a portion of a printer driver routine that sets up a default printing view device. It first sets up a bitmap structure with default values, and then creates a bitmap shape to pass to `GXNewViewDevice`. The view group for this view device is `gxScreenViewDevices`, because it is to be used for printing, not offscreen drawing.

Listing 7-9 Creating a new view device

```

gxBitmap      aBitmap;
gxViewDevice  vd;
aBitmap.pixelSize = 1;
aBitmap.rowBytes = 0;
aBitmap.width   = 0;
aBitmap.height  = 0;
aBitmap.image   = nil;
aBitmap.space   = gxNoSpace;
aBitmap.set     = nil;
aBitmap.profile = nil;

theBitmap = GXNewBitmap(&aBitmap, nil);
/* error-check here (not shown) */
.
.
vd = GXNewViewDevice(gxScreenViewDevices, theBitmap);
/* error-check here (not shown) */
.
.

```

When the driver is finished with the view device, it disposes of it with this line:

```
GXDisposeViewDevice(vd);
```

Listing 7-10 demonstrates creating a new view device by using the `GXCopyToViewDevice` function. Like **Listing 7-2** on page 7-44, it is part of a routine that creates an offscreen view group (`newGroup`) that is a copy of an existing view group (`group`). For each view device in the onscreen view group, the routine creates a copy and assigns it to the offscreen view group.

The routine decrements the `count` variable to control incrementing through the list of view devices (`list`) belonging to the onscreen view group.

Listing 7-10 Copying the view devices from one view group to another

```

long          deviceCount = GXGetViewGroupViewDevices(group, nil);
long          count = deviceCount;
gxViewDevice *deviceList = (void *)NewPtr(deviceCount *
                                         sizeof(gxViewDevice));
gxViewDevice *list = deviceList;
GXGetViewGroupViewDevices(group, deviceList);
while (count-- > 0)
{
    GXSetViewDeviceViewGroup(*list = GXCopyToViewDevice(nil,
                                                         *list), newGroup);
    list++;
}

```

The `GXNewViewDevice` function is described on page 7-98. The

`GXDisposeViewDevice` function is described on page 7-99.

The `GXCopyToViewDevice` function is described on page 7-100. The

`GXEqualViewDevice` function is described on page 7-101.

Manipulating View Device Object Properties

This section describes how to manipulate the bitmap, view group, and attributes properties of a view device object:

- n To manipulate the bitmap structure, you use the functions `GXGetViewDeviceBitmap` and `GXSetViewDeviceBitmap`.
- n To manipulate the view group reference, you use the functions `GXGetViewDeviceViewGroup` and `GXSetViewDeviceViewGroup`.
- n To manipulate the view device attributes, you use the functions `GXGetViewDeviceAttributes` and `GXSetViewDeviceAttributes`.
- n To manipulate the view device tag list, you use the functions `GXGetViewDeviceTags` and `GXSetViewDeviceTags`.

How to manipulate other view device properties is described in subsequent sections, starting with “Getting and Setting a View Device’s Clip and Mapping” on page 7-56.

Getting and Setting a View Device's Bitmap

The following code fragment is a function that uses `GXGetViewDeviceBitmap` to gain access to a copy of the color set of a view device, clone its reference (so it won't be deleted when its bitmap is disposed of), and then return it as a function result. The function needs the bitmap shape itself only temporarily, and therefore disposes of it after extracting the color set reference from it.

```
gxColorSet GetViewDeviceColorSet(gxViewDevice source)
{
    register gxShape    bitmapShape =
                                GXGetViewDeviceBitmap(source);
    register gxColorSet result = GetShapeColorSet(bitmapShape);
    if (result)
        GXCloneColorSet(result);
    GXDisposeShape(bitmapShape);
    return result;
}
```

The following code fragment is a function that uses `GXSetViewDeviceBitmap` to assign a color profile to a view device. The function disposes of its reference to the bitmap shape after assigning it to the view device.

```
void SetViewDeviceColorProfile(gxViewDevice target,
                               gxColorProfile profile)
{
    register gxShape bitmapShape = GXGetViewDeviceBitmap(target);

    SetShapeColorProfile(bitmapShape, profile);
    GXSetViewDeviceBitmap(target, bitmapShape);
    GXDisposeShape(bitmapShape);
}
```

The `GXGetViewDeviceBitmap` function is described on page 7-107; the `GXSetViewDeviceBitmap` function is described on page 7-108.

Getting and Setting a View Device's View Group

You can use the `GXGetViewDeviceViewGroup` function to retrieve the view group that a view device belongs to, and you can use the `GXSetViewDeviceViewGroup` to change the view group of a view device. Listing 7-10 on page 7-54 shows an example of using `GXSetViewDeviceViewGroup` to reassign the copy of a view device from one view group to another.

The `GXGetViewDeviceViewGroup` function is described on page 7-109; the `GXSetViewDeviceViewGroup` function is described on page 7-109.

Getting and Setting a View Device's Attributes and Tag References

You can examine the attributes of a view device object using the `GXGetViewDeviceAttributes` function. You can set the attributes of a view device object using the `GXSetViewDeviceAttributes` function. By setting attributes, you can influence whether the device bitmap is placed on an accelerator card and whether the device is active or inactive.

You can examine the list of references to tag objects currently associated with a view device object using the `GXGetViewDeviceTags` function. Once you create a tag object, you can attach it to a view device object using the `GXSetViewDeviceTags` function. You can attach as many tag objects as you like to a view device object.

Tag objects and the basic functions for manipulating them are described in the chapter "Tag Objects" in this book. That chapter also lists the common tag types defined and reserved by Apple Computer, Inc.

The `GXGetViewDeviceAttributes` function is described on page 7-110; the `GXSetViewDeviceAttributes` function is described on page 7-111. View device attributes are described in the section "View Device Attributes" on page 7-27.

The `GXGetViewDeviceTags` function is described on page 7-112. The `GXSetViewDeviceTags` function is described on page 7-113.

Getting and Setting a View Device's Clip and Mapping

The clip and mapping properties of a view device control its active imaging area, its scale (pixel size), and its position in global space. For onscreen view devices and printing view devices, you cannot change the clip or mapping properties; they are set by QuickDraw GX. For offscreen view devices, you can set the clip and mapping yourself. The functions you use are `GXGetViewDeviceClip`, `GXSetViewDeviceClip`, `GXGetViewDeviceMapping`, and `GXSetViewDeviceMapping`.

Listing 7-11 is a utility routine that returns a mapping matrix that converts from local space (or from the identity mapping, if the view port is `nil`) to device space (or to global space, if the view device is `nil`). It uses `GXGetViewDeviceMapping` to retrieve the view device's mapping matrix. (If the view device is `nil`, the routine uses the mapping matrix returned by the `GXGetViewPortGlobalMapping` function.) The routine also makes use of the `MapMapping` function, described in the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

Listing 7-11 Returning the mapping from local to device space

```

static void GetSpaceMapping(gxViewPort port, gxViewDevice device,
                           gxMapping *map)
{
    if (port)
        GXGetViewPortGlobalMapping(port, map);
    else
        ResetMapping(map);

    if(device)
    {
        gxMapping temp;
        MapMapping(map, GXGetViewDeviceMapping(device, &temp));
    }
}

```

The following code fragment is part of a printer driver routine that sets up a default view device. This section of code rescales the mapping matrix (`vdMapping`) of the view device (`vd`) from the default resolution (72 ppi, as specified by the identity matrix) to the horizontal and vertical resolution of the printer (`kHorizHighRes` and `kVertHighRes`). To do the scaling, the code uses the `ScaleMapping` function, described in the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*. It then uses `GXSetViewDeviceMapping` to assign the scaled mapping to the view device.

```

Fixed    xScale;
Fixed    yScale;

xScale = FixRatio(kHorizHighRes, 72);
yScale = FixRatio(kVertHighRes, 72);

ResetMapping(&vdMapping);
ScaleMapping(&vdMapping, xScale, yScale, ff(0), ff(0));
GXSetViewDeviceMapping(vd, &vdMapping);

```

The `GXGetViewDeviceClip` function is described on page 7-102; the `GXSetViewDeviceClip` function is described on page 7-103.

The `GXGetViewDeviceMapping` function is described on page 7-105; the `GXSetViewDeviceMapping` function is described on page 7-106.

Identifying a Shape's View Devices

The `GXGetShapeGlobalViewDevices` function returns a list of all view devices that a shape would actually appear in if it were drawn. The function can test the shape against all the shape's view ports, or you can specify a single view port for the test.

You can use `GXGetShapeGlobalViewDevices` to avoid the overhead of testing the drawing characteristics (such as the colors) of shapes on devices that they cannot be drawn to.

Listing 7-12 is part of a library routine that sets up a data structure for offscreen drawing through a given view port. This part of the code creates a full shape—which covers all of coordinate space—and passes it to `GXGetShapeGlobalViewDevices` to derive a count of all view devices that could be drawn on through the given port. (Note that, for the purpose of retrieving all the view devices of a view port, you could also use the `GXGetViewPortViewDevices` function.)

Listing 7-12 Setting up a data structure for offscreen drawing

```
viewPortBuffer NewViewPortBuffer(gxViewPort port)
{
    viewPortBuffer          buffersHandle;
    viewPortBufferRecord    *buffers;
    gxTransform             xform;
    gxShape                 area;
    long                   deviceCount;
    short                  i;

    NilParamReturnNil(port);          /* error check port parameter */
    area = GXNewShape(gxFullType);
    xform = GXNewTransform();
    GXSetTransformViewPorts(xform, 1, &port);
    GXSetShapeTransform(area, xform);
    GXDisposeTransform(xform);
    deviceCount = GXGetShapeGlobalViewDevices(area, port, nil);
    .
    . /* continued as Listing 7-13 on page 7-61 */
    .
}
```

The `GXGetShapeGlobalViewDevices` function is described on page 7-115.

Measuring a Shape in Device Space

You can use view device functions to measure a shape on a device in three ways:

- n The `GXGetShapeDeviceBounds` function measures the position and size of the bounding rectangle of a shape on a device, in device coordinates.
- n The `GXGetShapeDeviceArea` function determines the area of a shape (in pixels) on a device.
- n The `GXGetShapeDeviceColors` function determines the colors with which a shape would be drawn on a device.

The `GXGetShapeDeviceBounds` function determines whether any part of a shape intersects a view device, and if so, returns the bounding rectangle of that part of the shape in device coordinates. You can thus use `GXGetShapeDeviceBounds` to measure the size of a shape on a view device and to compare it with other shapes on the device. (To measure a shape in the local space of a view port, you can use the `GXGetShapeLocalBounds` function; to measure a shape in the global space of a view group, use `GXGetShapeGlobalBounds`.)

The following is a fragment of a function that converts a QuickDraw GX shape on a device into a QuickDraw picture. It uses `GXGetShapeDeviceBounds` to get the shape's bounding rectangle on the device, converts that rectangle into a QuickDraw `rect` structure, further converts it to QuickDraw local coordinates, and uses that to define the picture bounding rectangle. After that, the function converts the shape itself (not shown).

```

GXGetShapeDeviceBounds(theShape, 0, 0, &shapeBounds);
picRect.left          = FixedToInt(shapeBounds.left);
picRect.top           = FixedToInt(shapeBounds.top);
picRect.right         = FixedToInt(shapeBounds.right);
picRect.bottom        = FixedToInt(shapeBounds.bottom);
GlobalToLocal((Point*) &picRect.top);
GlobalToLocal((Point*) &picRect.bottom);

thePicture = OpenPicture(&picRect);
.
. /* convert the shape (not shown) */
.

```

The QuickDraw functions `GlobalToLocal` and `OpenPicture`, and the data types `Point` and `Rect` are described in *Inside Macintosh: Imaging With QuickDraw*.

Note

You do not need to write special functions to convert QuickDraw pictures into QuickDraw GX shapes. You can use the QuickDraw-to-QuickDraw GX translator for that; see the Macintosh environment chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*. u

View-Related Objects

The `GXGetShapeDeviceBounds` function is described on page 7-116. The

`GXGetShapeDeviceArea` function is described on page 7-118.

The `GXGetShapeDeviceColors` function is described on page 7-119.

`GXGetShapeGlobalBounds` function is described on page 7-125.

The `GXGetShapeLocalBounds` function, described on page 7-96.

Hit-Testing a Shape on a Device

The `GXHitTestDevice` function is one of several hit-testing functions provided by QuickDraw GX. Hit-testing in general is described in the chapter “Introduction to QuickDraw GX” in this book; shape parts for hit-testing are described in the chapter “Transform Objects” in this book.

You use `GXHitTestDevice` instead of `GXHitTestShape` or `GXHitTestPicture`—or before them—when it is important to take into account whether a shape is actually visible on a device. Unlike `GXHitTestShape` and `GXHitTestPicture`, `GXHitTestDevice` accounts for clipping and does not return successful hits for shapes that are not actually drawn.

Another significant difference is that the tolerance for `GXHitTestDevice` defines a rectangular area of pixels, not the circular geometry area used by `GXHitTestShape` and `GXHitTestPicture`. Thus you can use the tolerance value for `GXHitTestDevice` as something like a clip area, expanding it to cover an entire window or contracting it to one or a few complete pixels.

What `GXHitTestDevice` does not do that `GXHitTestShape` and `GXHitTestPicture` do is analyze the parts of a shape. If you are hit-testing in order to highlight specific parts of a shape, for example, you can first call `GXHitTestDevice` to determine which shape was actually hit, and then call `GXHitTestShape` or `GXHitTestPicture` to determine the part or parts to highlight.

The `GXHitTestDevice` function is described on page 7-120. The `GXHitTestShape` function is described in the chapter “Shape Objects” in this book. The `GXHitTestPicture` function is described in the picture shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*.

Using View Groups

This section demonstrates how to use QuickDraw GX view groups. It shows how you can

- n create and manipulate view group objects
- n set up an offscreen drawing environment
- n measure a shape in a view group

Creating and Manipulating View Group Objects

QuickDraw GX provides the `GXNewViewGroup` function to allow you to create a new view group object, and the `GXDisposeViewGroup` function to delete it. Normally, you create a view group only for the purpose of offscreen drawing.

Listing 7-13 is a continuation of the routine in Listing 7-12 on page 7-58 that sets up a data structure for offscreen drawing through a given view port. This part of the code fills in various fields of the `buffers` data structure and calls `GXNewViewGroup` to create an offscreen view group. In addition, it calls `GXNewViewPort` to create an offscreen view port (`slavePort`) in the new group, and `GXGetShapeGlobalViewDevices` to copy all drawable devices from the original onscreen view port (`port`) into a list in the data structure.

Listing 7-13 Setting up a data structure for offscreen drawing

```
.
.  /* continued from Listing 7-12 on page 7-58 */
.
    buffersHandle = (viewPortBuffer) NewHandle(sizeof
                (viewPortBufferRecord) + (deviceCount - 1) *
                sizeof(gxViewDevice));

    NilParamReturnNil(port);      /* error-check the handle */
    HLock((Handle) buffersHandle);
    buffers = *buffersHandle;
    buffers->group = GXNewViewGroup();
    buffers->masterPort = port;
    buffers->slavePort = GXNewViewPort(buffers->group);
    buffers->area = area;
    buffers->draw = GXNewShape(gxPictureType);
    buffers->deviceCount = deviceCount;
    GXSetViewPortDither(buffers->slavePort,
                GXGetViewPortDither(port));
    GXGetShapeGlobalViewDevices(area, port, buffers->devices);
```

Once you have created a view group object, you can assign view ports and view devices to it with the `GXSetViewPortViewGroup` function (as described under “Manipulating View Port Object Properties” beginning on page 7-42) and the `GXSetViewDeviceViewGroup` function (as described under “Manipulating View Device Object Properties” beginning on page 7-54).

At any time, you can retrieve a list of view ports or view devices that belong to a view group by calling the functions `GXGetViewGroupViewPorts` and `GXGetViewGroupViewDevices`. See Listing 7-2 on page 7-44 for an example of the use of `GXGetViewGroupViewPorts`; see Listing 7-10 on page 7-54 for an example of the use of `GXGetViewGroupViewDevices`.

When you have finished using an offscreen view group, you delete it with the `GXDisposeViewGroup` function. For an example of the use of `GXDisposeViewGroup`, see page 7-63.

The `GXNewViewGroup` function is described on page 7-122. The `GXDisposeViewGroup` function is described on page 7-122.

The `GXGetViewGroupViewPorts` function is described on page 7-123. The `GXGetViewGroupViewDevices` function is described on page 7-124.

Setting Up an Offscreen View Group

This section shows how to set up a view port for offscreen drawing. Examples of most of the steps shown here have already been presented elsewhere in this chapter, although not all together.

Offscreen drawing requires creating a new view group, so that drawing does not conflict with the screen devices view group. You must also create a view port to draw into. To set up the view port, you must create a view device object; however, for offscreen drawing the view device object need not correspond to any physical device. Follow this typical sequence of steps to set up an offscreen view group:

1. Create a new view group.
2. Create a new view device in this view group, specifying a bit map that represents the area that you may want to copy onscreen later.
3. Create a new view port in this view group.
4. Retrieve the bitmap as a shape object of its own, so that you can later draw it directly onscreen.
5. Create a transform object for your shapes, and assign the view port to its view port list.

Listing 7-14 is a partial example of a routine that sets up an offscreen drawing environment. It does not show how the bitmap shape (`bitShape`) for the device is set up. See the bitmap shapes chapter of *Inside Macintosh: QuickDraw GX Graphics* for information on bitmap shapes.

Listing 7-14 Setting up a view port and view group for offscreen drawing

```

gxShape      myDraw, bitShape;
gxTransform  myXform;
gxViewDevice myDevice;
gxViewPort   myPort;
gxViewGroup  myGroup;
.
. /* set up bitmap shape for offscreen view device */
.
myGroup = GXNewViewGroup();
myDevice = GXNewViewDevice(myGroup, bitShape);
myPort = GXNewViewPort(myGroup);
myDraw = GXGetViewDeviceBitmap(myDevice);
myXform = GXNewTransform();
GXSetTransformViewPorts(myXform, 1, &myPort);

```

The `myDraw` shape represents your offscreen drawing buffer. Whenever you draw a shape that has `myXform` as its transform object, drawing takes place offscreen, in the pixel image associated with `myDraw`. If you added a view port in the onscreen view group to the view port list of `myXform`, drawing could take place both offscreen and onscreen simultaneously.

It is useful to have a direct reference to `myDraw` because you can draw it itself, which would have the effect of transferring the offscreen buffer onto the screen (if `myDraw` references a transform object that references onscreen view ports).

When you are finished with offscreen drawing, you can dispose of the objects you have created:

```

GXDisposeShape(myDraw);
GXDisposeTransform(myXform);
GXDisposeViewGroup(myGroup);

```

You do not have to explicitly dispose of your offscreen view port or view device, because calling `GXDisposeViewGroup` causes QuickDraw GX to dispose of all of its view ports and view devices.

Measuring a Shape in Global Space

The `GXGetShapeGlobalBounds` function measures the bounding rectangle of a shape in global coordinates—that is, after the transform mapping has been applied to the shape geometry, and after all view port mappings have been applied. You can thus use `GXGetShapeGlobalBounds` to compare the true positions and sizes of any two shapes in the same view group, even if they do not share the same view port or do not appear on the same view device. (To compare the positions and sizes of two shapes in the same view port, you can use the `GXGetShapeLocalBounds` function; to measure a shape on a view device, use `GXGetShapeDeviceBounds`.)

View-Related Objects

Listing 7-15 is a library function used in offscreen drawing. The function returns the device characteristics (a bitmap structure plus an offset) of a particular “area,” the intersection of an offscreen view device and an offscreen view port. Area-characteristics structures are used by this library to store device-specific drawing information for each area within the offscreen view port occupied by the pixel image of a device. This listing uses `GXGetShapeGlobalBounds` to determine the intersection of the specified shape (area) with the specified view device (device), which determines the size of the image stored in the area-characteristics structure (x).

Listing 7-15 Returning the characteristics of an offscreen device area

```
static areaCharacteristics GetAreaCharacteristics(gxShape area,
                                                gxViewDevice device,
                                                gxViewPort port)
{
    areaCharacteristics x; /* a bitmap structure & location */
    gxRectangle bounds;
    gxShape bitShape;
    gxMapping map;
    .
    .
    . /* get device bitmap and shape bounds on device */
    bitShape = GXGetViewDeviceBitmap(device);
    GXGetShapeGlobalBounds(area, port, nil, &bounds);

    /* fill out the area-characteristics structure */
    GXGetBitmap(bitShape, &x.bits, nil);
    if (x.bits.space == gxIndexedSpace)
        GXCloneColorSet(x.bits.set);
    if (x.bits.profile)
        GXCloneColorProfile(x.bits.profile);
    GXDisposeShape(bitShape);
    x.offset.x = bounds.left;
    x.offset.y = bounds.top;
    x.bits.width = FixedRound(bounds.right) -
                  FixedRound(bounds.left);
    x.bits.height = FixedRound(bounds.bottom) -
                   FixedRound(bounds.top);

    /* map the area offset back to local space, store in x */
    InvertMapping(&map, GXGetViewPortGlobalMapping(port, &map));
    MapPoints(&map, 1, &x.offset);
    return x;
}
```

The `GXGetShapeGlobalBounds` function is described on page 7-125.

The `GXGetShapeLocalBounds` function is described on page 7-96. The `GXGetShapeDeviceBounds` function is described on page 7-116.

View-Related Objects Reference

This section provides reference information to the structures and functions that allow you to create and manipulate view related objects and alter their properties. It includes

- n descriptions of the constants and data types that are specific to view port, view device, and view group objects
- n descriptions of the QuickDraw GX functions that operate on view port objects
- n descriptions of the QuickDraw GX functions that operate on view device objects
- n descriptions of the QuickDraw GX functions that operate on view group objects

Constants and Data Types

This section describes the data types that you use to obtain and provide information about view port, view device, and view group objects.

The View Port Object

QuickDraw GX provides you with access to an individual view port object through a `gxViewPort` reference:

```
typedef struct gxPrivateViewPortRecord *gxViewPort;
```

In this type definition, `gxViewPort` is a type-checked reference, not an actual pointer to any defined structure. The contents of the view port object are private.

The Halftone Structure

Halftones are described by the `gxHalftone` structure:

```
struct gxHalftone{
    Fixed          angle;
    Fixed          frequency;
    gxDotType      method;
    gxTintType     tinting;
    gxColor        dotColor;
    gxColor        backgroundColor;
    gxColorSpace   tintSpace;
};
```

Field descriptions

angle	The orientation of the rows of dots in the halftone pattern. It is a fixed-point number between 0.0 and 360.0 that describes an angle, in degrees, clockwise from horizontal.
frequency	The size of the cells, in terms of numbers of dots per inch. It can be any positive value.
method	The halftone pattern itself and how it is filled: the shapes of the dots, the pattern of their arrangement, and the way in which a dot fills its cell as it enlarges. The supported methods are defined in the <code>gxDotTypes</code> enumeration, described next.
tinting	The type of calculation by which the input color is to be approximated by a ratio of dot color and background color. Tint types are defined in the <code>gxTintTypes</code> enumeration, described on page 7-67.
dotColor	The color of the dots used to form the halftone.
backgroundColor	The color of the background used to form the halftone.
tintSpace	The color space the input color is converted to before the halftone calculations are made.

The halftone structure is described further in the section “Halftone” beginning on page 7-13.

Dot Types

The `gxDotTypes` enumeration defines the possible halftone dot types:

```
enum gxDotTypes{
    gxRoundDot = 1,
    gxSpiralDot,
    gxSquareDot,
    gxLineDot,
    gxEllipticDot,
    gxTriangleDot,
    gxDispersedDot
};

typedef long gxDotType;
```

The meaning of each dot type is evident from its name, except perhaps for `gxDispersedDot`. The `gxDispersedDot` dot type uses a seemingly random pattern of small dots that gradually fill up each cell as the tint value increases. For a visual representation of each of these dot types, see Figure 7-6 on page 7-16.

Tint Types

The `gxTintTypes` enumeration defines the possible ways of calculating the tint color (the color to be represented by a ratio of dot and background color) for a halftone.

```
enum gxTintTypes{
    gxNoTint,
    gxLuminanceTint,
    gxAverageTint,
    gxMixtureTint,
    gxComponent1Tint,
    gxComponent2Tint,
    gxComponent3Tint,
    gxComponent4Tint
};

typedef long gxTintType;
```

Constant descriptions

`gxNoTint` No tint color. In a halftone structure with all fields set to 0, the tinting field has a value of `gxNoTint`.

`gxLuminanceTint` The tint color is the input color's luminance.

`gxAverageTint` The tint color is the average of all components of the input color.

`gxMixtureTint` Project the input color onto the foreground-background color axis in tint color space. That projection point is the tint color.

`gxComponent1Tint` Use only component 1 of the input color as the tint color.

`gxComponent2Tint` Use only component 2 of the input color as the tint color.

`gxComponent3Tint` Use only component 3 of the input color as the tint color.

`gxComponent4Tint` Use only component 4 of the input color as the tint color.

For more information about halftone tints, see the section "Halftone" beginning on page 7-13.

View Port Attributes

The view port attributes are a set of flags that modify the behavior of the view port object. Constants for all recognized view port attributes are defined in the `gxPortAttributes` enumeration:

```
enum gxPortAttributes {
    gxGrayPort          = 0x0001,    /* convert to gray space */
    gxAlwaysGridPort    = 0x0002,    /* use gxDeviceGridStyle */
    gxEnableMatchPort   = 0x0004    /* perform color matching */
};

typedef long gxPortAttribute;
```

The individual view port attributes are described in Table 7-2 on page 7-20.

The View Device Object

QuickDraw GX provides you with access to an individual view device object through a `gxViewDevice` reference:

```
typedef struct gxPrivateViewDeviceRecord *gxViewDevice;
```

In this type definition, `gxViewDevice` is a type-checked reference, not an actual pointer to any defined structure. The contents of the view device object are private.

View Device Attributes

The view device attributes are a set of flags that modify the behavior of the view device object. Constants for all recognized view device attributes are defined in the `gxDeviceAttributes` enumeration:

```
enum gxDeviceAttributes{
    gxDirectDevice      = 0x01,    /* pixel image must be accessible */
    gxRemoteDevice      = 0x02,    /* pixel image may be on card */
    gxInactiveDevice    = 0x04    /* device is inactive */
};

typedef long gxDeviceAttribute;
```

The individual view device attributes are described in Table 7-3 on page 7-27.

The View Group Object

QuickDraw GX provides you with access to an individual view group object through a `gxViewGroup` reference:

```
typedef struct gxPrivateViewGroupRecord *gxViewGroup;
```

In this type definition, `gxViewGroup` is a type-checked reference, not an actual pointer to any defined structure. The contents of the view group object are private.

View Group Types

QuickDraw GX provides two predefined view group references for you, defined by the following constants:

```
#define gxAllViewDevices          ((gxViewGroup) 0)
#define gxScreenViewDevices      ((gxViewGroup) 1)
```

Constant descriptions

`gxAllViewDevices`

Not an actual reference, this constant represents all view groups, both offscreen and onscreen. You can use this constant when you want to use the `GXGetViewGroupViewPorts` function or the `GXGetViewGroupViewDevices` function to determine all the view ports or all the view devices for all view groups. You cannot use this constant to set a view port or view device.

`gxScreenViewDevices`

A reference to the view group that includes view device objects for all physical display devices. Only by drawing to view ports in this view group can you perform onscreen drawing. This is the one view group that QuickDraw GX provides for you.

View Port Functions

This section describes the QuickDraw GX functions you use with view port objects. Using the functions described here, you can

- n create and manipulate view port objects
- n manipulate view port object properties
- n retrieve the view devices that intersect a view port
- n retrieve the view ports that intersect a shape
- n measure a shape in a view port

To associate a view port object directly with a QuickDraw GX transform object, use the `GXGetTransformViewPorts` and `GXSetTransformViewPorts` functions. To associate a view port object indirectly with a QuickDraw GX shape object, use the `GXGetShapeViewPorts` and `GXSetShapeViewPorts` functions. All four functions are described in the chapter “Transform Objects” in this book.

Creating and Manipulating View Port Objects

The functions described in this section allow you to create and manipulate view port objects. With the functions in this section, you can

- n create and dispose of view ports
- n copy view ports
- n test view ports for equality

GXNewViewPort

You can use the `GXNewViewPort` function to create a new view port.

```
gxViewPort GXNewViewPort(gxViewGroup group);
```

`group` A reference to the view group in which to create the view port.

function result A reference to the newly created view port.

DESCRIPTION

The `GXNewViewPort` function creates a new view port with default properties and assigns it to the specified view group. All other properties are set to their default values:

- n no parent view port or child view port
- n a clip that is a full shape
- n a mapping that is the identity mapping
- n a dither level of 1
- n no halftone
- n no attributes set
- n an empty tag list

To create a view port in the onscreen view group, pass the value `gxScreenViewDevices` for the `group` parameter. To obtain an offscreen view group reference to pass to this function, use the `GXNewViewGroup` function.

SPECIAL CONSIDERATIONS

If no error occurs, the `GXNewViewPort` function creates a view port object; you are responsible for disposing of that object when you no longer need it.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`invalid_viewGroup_reference`

SEE ALSO

For examples of the use of this function, see page 7-41, Listing 7-5 on page 7-47, and Listing 7-14 on page 7-63.

To dispose of a view port, use the `GXDisposeViewPort` function, described next. To dispose of all the view ports in a view group, use the `GXDisposeViewGroup` function, described on page 7-122.

The `gxScreenViewDevices` view group reference is described in the section “View Group Types” on page 7-69. The `GXNewViewGroup` function is described on page 7-122.

GXDisposeViewPort

You can use the `GXDisposeViewPort` function to delete a view port object.

```
void GXDisposeViewPort(gxViewPort target);
```

`target` A reference to the view port object to dispose of.

DESCRIPTION

The `GXDisposeViewPort` function disposes of the target view port. If the target view port is a parent, its children are removed from their position in the hierarchy and each is made the root of a new hierarchy.

SPECIAL CONSIDERATIONS

If the target view port is a parent associated with a window, its child view ports lose their association with the window.

ERRORS, WARNINGS, AND NOTICES**Errors**

`invalid_viewPort_reference`

SEE ALSO

For an example of the use of this function, see page 7-41.

For information about view port hierarchies, see “Parent and Child View Ports” beginning on page 7-18.

GXCopyToViewPort

You can use the `GXCopyToViewPort` function to create a copy of an existing view port object.

```
gxViewPort GXCopyToViewPort(gxViewPort target, gxViewPort source);
```

`target` A reference to the view port to copy the source contents into. If you specify `nil` for this parameter, the `GXCopyToViewPort` function creates a new view port object.

`source` A reference to the view port whose contents you want to copy.

function result A reference to the copy (that is, the target view port) of the source view port.

DESCRIPTION

The `GXCopyToViewPort` function copies the contents of an existing view port object to another, or it creates a new view port object and copies the contents of an existing view port object to it. The function copies the clip, mapping, dither, halftone, attributes, and tag list (but not the parent or child view ports) of the view port object specified by the `source` parameter into the view port object specified by the `target` parameter. It clones, but does not copy, the tag objects in the tag list. The target view port is placed in the same view group as the source view port. The target view port is not associated with any window, whether or not the source view port is associated with one.

If you specify `nil` for the `target` parameter, the `GXCopyToViewPort` function creates a new view port object and copies the source properties into it.

You can use the `GXCopyToViewPort` function to create a copy of a view port object and then modify it without changing the original.

SPECIAL CONSIDERATIONS

If you specify `nil` for the `target` parameter and no error occurs, the `GXCopyToViewPort` function creates a view port object; you are responsible for disposing of that object when you no longer need it.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`invalid_viewPort_reference`
`viewPort_is_a_window` (debugging version)

SEE ALSO

For an example of the use of this function, see Listing 7-2 on page 7-44.

To create a new view port that has default values instead of being a copy of an existing view port, use the `GXNewViewPort` function, described on page 7-70.

To compare two view port objects for equality, use the `GXEqualViewPort` function, described in the next section.

GXEqualViewPort

You can use the `GXEqualViewPort` function to determine whether two view port objects are equal.

```
boolean GXEqualViewPort(gxViewPort one, gxViewPort two);
```

`one` A reference to a view port to test for equality.

`two` A reference to another view port to test for equality.

function result `true` if the view port specified by the `one` parameter is equal to the view port specified by the `two` parameter; `false` otherwise.

DESCRIPTION

The `GXEqualViewPort` function returns as its function result a Boolean value indicating whether the view port object specified by the `one` parameter is equal to the view port object specified by the `two` parameter.

For two view port objects to be equal, they must have identical mappings, clips, dithers, halftones, and attributes. They also must have the same parent view port, if any, and (therefore) be in the same view group. If one view port is attached to a window, the other view port must be attached to the same window. The tag lists or child view ports of the view ports need not be identical.

ERRORS, WARNINGS, AND NOTICES**Errors**

`invalid_viewPort_reference`

SEE ALSO

To make a copy of a view port object that is equal by the criteria of this function, use the `GXCopyToViewPort` function, described in the previous section.

Manipulating View Port Object Properties

The functions described in this section allow you to manipulate the properties of the view port object. With these functions you can

- n get and set a view port's clip
- n get and set a view port's mapping, and get its global mapping
- n get and set a view port's dither and halftone, and get its halftone device angle
- n get and set a view port's parent view port and list of child view ports
- n get and set a view port's view group
- n get and set a view port's attributes and tag list

GXGetViewPortClip

You can use the `GXGetViewPortClip` function to examine the clip property of a view port object.

```
gxShape GXGetViewPortClip(gxViewPort source);
```

source A reference to the view port whose clip you wish to examine.

function result A reference to a newly created shape object that is a copy of the source view port's clip.

DESCRIPTION

The `GXGetViewPortClip` function returns a shape object whose geometry defines the clip associated with the view port. The function returns `nil` if there is no clip. The clip shape is a copy of the view port's clip; changing this shape does not change the view port's clip.

SPECIAL CONSIDERATIONS

If no error occurs, the `GXGetViewPortClip` function creates a shape object; you are responsible for disposing of that object when you no longer need it.

ERRORS, WARNINGS, AND NOTICES**Errors**

out_of_memory
invalid_viewPort_reference

SEE ALSO

For an example of the use of this function, see Listing 7-3 on page 7-45.

To set the view port's clip, use the `GXSetViewPortClip` function, described next.

GXSetViewPortClip

You can use the `GXSetViewPortClip` function to set the clip property of a view port object.

```
void GXSetViewPortClip(gxViewPort target, gxShape clip);
```

<code>target</code>	A reference to the view port whose clip you wish to set.
<code>clip</code>	A reference to a shape object whose geometry describes the clip to be assigned.

DESCRIPTION

The `GXSetViewPortClip` function copies information from the shape object referenced by the `clip` parameter into the clip property of the view port object referenced by the `target` parameter. You can specify `nil` for the `clip` parameter, in which case this function sets the clip property of the target view port to a full clip. (A full clip indicates that QuickDraw GX should not apply view port clipping to shapes drawn to this view port.)

Although a filled rectangle shape is most typical for a view port clip, the new clip shape may be a geometric shape, a bitmap shape, or a glyph shape. It may not be a picture, text, or layout shape.

- n If you specify a geometric shape, it must be in primitive form—that is, all the stylistic information about the shape must be incorporated into the shape's geometry—because this function copies only the geometry-related information from the shape you specify. It does not copy the information contained in the shape's style. You can convert a shape to its primitive form using the `GXPrimitiveShape` function, which is described in *Inside Macintosh: QuickDraw GX Graphics*. You can also specify an empty or full shape for a clip.

View-Related Objects

- n If you specify a bitmap shape, it must have a pixel size of 1 and its color profile reference must be `nil`. In the bitmap, pixel values of 0 obscure drawing; pixel values of 1 do not restrict visibility. The `GXSetViewPortClip` function copies the pixel image from the bitmap to the clip property of the target view port.
- n If you specify a glyph shape, this function uses information from the glyph shape's style object as well as its style list to determine the size, form, and position of the glyph outlines; those outlines are then used to clip drawing. The style list cannot have `nil` entries. A style object referenced by the glyph shape cannot be complex—that is, it cannot have a cap, join, dash, pattern, text face, font variation, tag list, or any of the properties used only by layout shapes.

Because it is copied into the view port, changing the clip shape after calling `GXSetViewPortClip` does not affect the view port's clip.

ERRORS, WARNINGS, AND NOTICES

Errors

<code>out_of_memory</code>	
<code>invalid_viewPort_reference</code>	
<code>colorProfile_must_be_nil</code>	(debugging version)
<code>bitmap_pixel_size_must_be_1</code>	(debugging version)
<code>empty_shape_not_allowed</code>	(debugging version)
<code>ignorePlatformShape_not_allowed</code>	(debugging version)
<code>nil_style_in_glyph_not_allowed</code>	(debugging version)
<code>complex_glyph_style_not_allowed</code>	(debugging version)
<code>illegal_type_for_shape</code>	(debugging version)
<code>shapeFill_not_allowed</code>	(debugging version)
<code>viewPort_is_a_window</code>	(debugging version)

Notices (debugging version)

`clip_already_set`
`tags_in_shape_ignored`

SEE ALSO

For examples of the use of this function, see Listing 7-4 on page 7-46 and Listing 7-5 on page 7-47.

For information about geometric shapes and bitmap shapes, see *Inside Macintosh: QuickDraw GX Graphics*. For information about glyph shapes, see *Inside Macintosh: QuickDraw GX Typography*.

To retrieve a copy of the view port clip, use the `GXGetViewPortClip` function, described in the previous section.

GXGetViewPortMapping

You can use the `GXGetViewPortMapping` function to retrieve the mapping for a view port object.

```
gxMapping *GXGetViewPortMapping(gxViewPort source,
                                gxMapping *map);
```

`source` A reference to the view port whose mapping you wish to examine.

`map` A pointer to a mapping structure. On return, this mapping contains a copy of the information from the mapping property of the source view port.

function result A pointer to a copy of the mapping property of the source view port. (This value is the same as the value returned in the `map` parameter.)

DESCRIPTION

The `GXGetViewPortMapping` function copies the mapping matrix information from the mapping property of the source view port object into the mapping structure pointed to by the `map` parameter. The function also returns as its function result a pointer to this mapping structure.

To make changes to the source view port's mapping property, you can alter the information returned by this function, and then use the `GXSetViewPortMapping` function to reassign the altered mapping to the source view port.

If the source view port is the root view port of a view port hierarchy, this function gives the same results as `GXGetViewPortGlobalMapping`.

ERRORS, WARNINGS, AND NOTICES

Errors

<code>invalid_viewPort_reference</code>	
<code>parameter_is_nil</code>	(debugging version)

SEE ALSO

For examples of the use of this function, see Listing 7-3 on page 7-45 and Listing 7-6 on page 7-48.

To set a view port's mapping, use the `GXSetViewPortMapping` function, described next.

For information about the `gxMapping` structure, see the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

GXSetViewPortMapping

You can use the `GXSetViewPortMapping` function to assign a mapping to a view port object.

```
void GXSetViewPortMapping(gxViewPort target,
                          const gxMapping *map);
```

`target` A reference to the view port object you want to assign the mapping to.
`map` A pointer to a mapping structure containing the mapping matrix to assign to the target view port.

DESCRIPTION

The `GXSetViewPortMapping` function copies information from the mapping structure pointed to by the `map` parameter into the mapping property of the target view port.

You can specify `nil` for the `map` parameter, in which case this function sets the mapping property of the target view port as follows:

- n If the clip shape is a full shape, which specifies no clipping, the function sets the mapping property to the identity mapping.
- n If a clip exists, the function sets the mapping's translation component to the upper-left corner of the clip. It sets the other components of the mapping to identity.

You can provide arbitrary values for the elements of the mapping structure pointed to by the `map` parameter, with one exception: the lower-right element of the matrix (element `[2][2]`) may not be 0.

ERRORS, WARNINGS, AND NOTICES

Errors

`invalid_viewPort_reference`
`viewPort_is_a_window` (debugging version)

SEE ALSO

For examples of the use of this function, see Listing 7-3 on page 7-45 and Listing 7-5 on page 7-47.

To get a view port's mapping, use the `GXGetViewPortMapping` function, described in the previous section.

For information about the `gxMapping` structure, see the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

GXGetViewPortGlobalMapping

You can use the `GXGetViewPortGlobalMapping` function to examine the resultant mapping after concatenating the mapping properties of a view port object with all its parent view ports.

```
gxMapping *GXGetViewPortGlobalMapping(gxViewPort source,
                                       gxMapping *map);
```

`source` A reference to the view port whose global mapping you wish to examine.

`map` A pointer to a mapping. On return, this parameter contains the concatenation of all the mapping properties of this view port and the view ports from which it descends.

function result A pointer to a copy of the `map` parameter. (This value is the same as the value returned in the `map` parameter.)

DESCRIPTION

The `GXGetViewPortGlobalMapping` function returns the global mapping of the source view port in its view group. This mapping is the result of concatenating the view port's mapping with that of its parent and that of its parent's parent, and so on; the concatenation ascends the view port's branch in the hierarchy, from its position to the root view port, inclusive.

This function is useful when you want to determine how a shape is mapped through the root view port into the view group; that is, what its position and dimensions are in global space.

If the source view port is the root view port of a view port hierarchy, this function gives the same results as `GXGetViewPortMapping`.

ERRORS, WARNINGS, AND NOTICES

Errors

`invalid_viewPort_reference`

SEE ALSO

For an example of the use of this function, see Listing 7-11 on page 7-57.

To get a view port's mapping without concatenating it with the mappings of parent view ports, use the `GXGetViewPortMapping` function, described on page 7-77.

For a discussion of coordinate systems, global space, and view port hierarchies, see the section "Global Space" beginning on page 7-34.

For information about the `gxMapping` structure, see the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

GXGetViewPortDither

You can use the `GXGetViewPortDither` function to examine the dither level of a view port object.

```
long GXGetViewPortDither(gxViewPort source);
```

`source` A reference to the view port whose dither level you wish to examine.

function result The view port's dither level.

DESCRIPTION

The `GXGetViewPortDither` function returns the view port dither level, which is a value between 0 and 16, inclusive. The values 0 and 1 both mean do not dither.

ERRORS, WARNINGS, AND NOTICES

Errors

`invalid_viewPort_reference`

SEE ALSO

For an example of the use of this function, see Listing 7-13 on page 7-61.

To set the dither level, use the `GXSetViewPortDither` function, described next.

For information about the dither property, see “Dither” beginning on page 7-10.

GXSetViewPortDither

You can use the `GXSetViewPortDither` function to assign a dither level to a view port object.

```
void GXSetViewPortDither(gxViewPort target, long level);
```

`target` A reference to the view port whose dither level you wish to set.

`level` The new dither level.

DESCRIPTION

The `GXSetViewPortDither` function specifies the default dither level for the target view port. You can specify a level in the range of 0 to 16, inclusive. Levels 0 and 1 specify no dithering, otherwise the level specifies the maximum number of pixels in the dither pattern.

SPECIAL CONSIDERATIONS

You can set the ink object's attribute, `gxSuppressDitherInk`, if you want to ignore the view port's dither level for shapes drawn with a specific ink. For more information about ink object attributes, see the chapter "Ink Objects" in this book.

Dithering does not occur on 32 bit-per-pixel devices, regardless of the dither level.

SPECIAL CONSIDERATIONS

Version 1.0 of QuickDraw GX does not guarantee useful results for dither levels greater than 4.

ERRORS, WARNINGS, AND NOTICES**Errors**

`invalid_viewPort_reference`
`parameter_out_of_range` (debugging version)

Notices (debugging version)

`dither_already_set`

SEE ALSO

For examples of the use of this function, see Listing 7-1 on page 7-42 and Listing 7-13 on page 7-61.

To get a view port's dither level, use the `GXGetViewPortDither` function, described in the previous section.

For information about the dither property, see "Dither" beginning on page 7-10.

GXGetViewPortHalftone

You can use the `GXGetViewPortHalftone` function to examine the halftone property of a view port object.

```
boolean GXGetViewPortHalftone(gxViewPort source,
                             gxHalftone *data);
```

`source` A reference to the view port whose halftone you wish to examine.

`data` A pointer to a halftone structure. On return, the structure contains the view port's halftone information.

function result true if the halftone exists; otherwise false.

DESCRIPTION

The `GXGetViewPortHalftone` function copies the view port's halftone into the structure you provide, pointed to by the `data` parameter. If a halftone does not exist, the contents of the structure pointed to by the `data` parameter are not changed.

ERRORS, WARNINGS, AND NOTICES**Errors**

`invalid_viewPort_reference`

SEE ALSO

For information about the halftone property, see “Halftone” beginning on page 7-13.

To set the halftone of a view port, use the `GXSetViewPortHalftone` function, described next.

GXSetViewPortHalftone

You can use the `GXSetViewPortHalftone` function to assign a halftone property to a view port object.

```
void GXSetViewPortHalftone(gxViewPort target,
                          const gxHalftone *data);
```

`target` A reference to the view port whose halftone you wish to change.
`data` A pointer to a halftone structure containing the data with which to set the view port's halftone property.

DESCRIPTION

The `GXSetViewPortHalftone` function sets the halftone for the target view port. If the `data` parameter is set to `nil`, halftones are not used when drawing to this view port.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>invalid_viewPort_reference</code>	
<code>frequency_parameter_out_of_range</code>	(debugging version)
<code>tinting_parameter_out_of_range</code>	(debugging version)
<code>method_parameter_out_of_range</code>	(debugging version)
<code>space_may_not_be_indexed</code>	(debugging version)
<code>colorSpace_out_of_range</code>	(debugging version)

Notices (debugging version)

`halftone_already_set`

SEE ALSO

For an example of the use of this function, see Listing 7-1 on page 7-42.

For information about the halftone property, see “Halftone” beginning on page 7-13.

To set the halftone of a view port, use the `GXGetViewPortHalftone` function, described in the previous section.

GXGetHalftoneDeviceAngle

You can use the `GXGetHalftoneDeviceAngle` function to determine the actual angle a halftone is drawn with on a particular view device.

```
Fixed GXGetHalftoneDeviceAngle(gxViewDevice source,
                               const gxHalftone *data);
```

`source` A reference to the view device whose halftone angle you wish to examine.
`data` A pointer to a halftone structure that specifies the characteristics of a halftone.

function result The halftone angle as it would be drawn on the view device.

DESCRIPTION

The `GXGetHalftoneDeviceAngle` function returns the actual angle that the specified halftone would be drawn with on the source view device. The contents of the halftone structure pointed to by the `data` parameter are not changed.

The halftone angle on the view device may be different from the one you set with the `GXSetViewPortHalftone` function because the view device’s resolution interacts with the halftone frequency, which specifies the dot size in cells per inch, and the view device’s grid pattern.

ERRORS, WARNINGS, AND NOTICES**Errors**

`invalid_viewDevice_reference`

SEE ALSO

For information about the halftone property, see “Halftone” beginning on page 7-13.

To get the halftone of a view port, use the `GXGetViewPortHalftone` function, described on page 7-81.

To obtain a list of view devices that intersect a view port, use the `GXGetViewPortViewDevices` function, described on page 7-94.

GXGetViewPortParent

You can use the `GXGetViewPortParent` function to retrieve the parent view port of a view port object.

```
gxViewPort GXGetViewPortParent(gxViewPort source);
```

`source` A reference to the view port whose parent you want to examine.

function result A reference to the source view port's parent view port.

DESCRIPTION

The `GXGetViewPortParent` function returns the parent view port of the source view port, or `nil` if the view port has no parent (that is, if it is the root view port in a hierarchy).

ERRORS, WARNINGS, AND NOTICES

Errors

`invalid_viewPort_reference`

SEE ALSO

For information about view port hierarchies and the parent view port property, see “Parent and Child View Ports” beginning on page 7-18.

To set the parent of a view port, use the `GXSetViewPortParent` function, described next.

To get the child view port list of a view port, use the `GXGetViewPortChildren` function, described on page 7-86. To set the child view port list of a view port, use the `GXSetViewPortChildren` function, described on page 7-87.

GXSetViewPortParent

You can use the `GXSetViewPortParent` function to assign a parent view port to a view port object.

```
void GXSetViewPortParent(gxViewPort target, gxViewPort parent);
```

`target` A reference to the view port whose parent you want to set.

`parent` A reference to the target view port's new parent.

DESCRIPTION

The `GXSetViewPortParent` function replaces the target's parent view port with the view port specified in the `parent` parameter. It also adds the target view port to the parent's list of child view ports. If the target view port is in a different view group from the new parent view port, the target view port is reassigned to the parent's view group, which also causes the target's child view ports to be assigned to the new parent's view group as well.

If you set the `parent` parameter to `nil`, this function sets the target view port to have no parent; the target view port then becomes the root of a new view port hierarchy.

SPECIAL CONSIDERATIONS

View port hierarchies cannot contain circular references; that is, a view port cannot have itself or any of its descendants as its parent.

You cannot assign a parent view port to a window view port (one attached to a Macintosh window).

The view ports in a hierarchy must all be in the same view group.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>invalid_viewPort_reference</code>	
<code>viewPort_is_a_window</code>	(debugging version)
<code>viewPort_cannot_contain_itself</code>	(debugging version)

SEE ALSO

For an example of the use of this function, see Listing 7-5 on page 7-47.

For information about view port hierarchies and the parent view port property, see "Parent and Child View Ports" beginning on page 7-18.

To get the parent of a view port, use the `GXGetViewPortParent` function, described in the previous section.

To get the child view port list of a view port, use the `GXGetViewPortChildren` function, described next. To set the child view port list of a view port, use the `GXSetViewPortChildren` function, described on page 7-87.

GXGetViewPortChildren

You can use the `GXGetViewPortChildren` function to retrieve the list of child view ports of a view port object.

```
long GXGetViewPortChildren(gxViewPort source, gxViewPort list[]);
```

`source` A reference to the view port whose child view ports you wish to examine.

`list` An array of view port references. On return, contains the child view port list of the source view port.

function result The number of references in the child view port list of the source view port.

DESCRIPTION

The `GXGetViewPortChildren` function retrieves the list of child view ports of the source view port. It also returns, as its function result, the number of references in the list. The list and the number returned by this function do not include children of children.

The view ports are placed in the list array in the order they were added as children.

If you set the `list` parameter to `nil`, `GXGetViewPortChildren` does not retrieve a list of references; it only returns the number of child view port objects. Therefore, you typically call this function twice: first to get the size of array to allocate for the `list` parameter, and second to retrieve the list itself.

ERRORS, WARNINGS, AND NOTICES

Errors

`invalid_viewPort_reference`

SEE ALSO

For information about view port hierarchies and the child view port list property, see “Parent and Child View Ports” beginning on page 7-18.

To set the child view port list of a view port, use the `GXSetViewPortChildren` function, described next.

To get the parent of a view port, use the `GXGetViewPortParent` function, described on page 7-84. To set the parent of a view port, use the `GXSetViewPortParent` function, described in the previous section.

GXSetViewPortChildren

You can use the `GXSetViewPortChildren` function to assign a list of child view ports to a view port object.

```
void GXSetViewPortChildren(gxViewPort target, long count,
                           const gxViewPort list[]);
```

<code>target</code>	A reference to the view port whose children you wish to set.
<code>count</code>	The number of references in the child view port list.
<code>list</code>	The array of view port references that is the new child view port list.

DESCRIPTION

The `GXSetViewPortChildren` function sets each of the view ports specified in the `list` parameter to have the target view port as its parent, and assigns the list as the child view port list of the target view port. Previous children of the target view port are assigned to have no parent.

If the `target` parameter is set to `nil`, each view port in the list is assigned to have no parent. If a child view port in the list is in a different view group than the target view port's view group, the child view port is changed to be in the view group of the target view port.

SPECIAL CONSIDERATIONS

View port hierarchies cannot contain circular references; that is, a view port cannot have itself or any of its ancestors as its child.

The view ports in a hierarchy must all be in the same view group.

ERRORS, WARNINGS, AND NOTICES

Errors

<code>invalid_viewPort_reference</code>	
<code>viewPort_cannot_contain_itself</code>	(debugging version)

SEE ALSO

For information about view port hierarchies and the child view port list property, see "Parent and Child View Ports" beginning on page 7-18.

To get the child view port list of a view port, use the `GXGetViewPortChildren` function, described in the previous section.

To get the parent of a view port, use the `GXGetViewPortParent` function, described on page 7-84. To set the parent of a view port, use the `GXSetViewPortParent` function, described on page 7-84.

GXGetViewPortViewGroup

You can use the `GXGetViewPortViewGroup` function to determine the view group that a view port is part of.

```
gxViewGroup GXGetViewPortViewGroup(gxViewPort source);
```

`source` A reference to the view port whose view group you wish to examine.

function result A reference to the view group that the source view port is part of.

DESCRIPTION

The `GXGetViewPortViewGroup` returns a reference to the source view port's view group. If it is the onscreen view group, the returned value is `gxScreenViewDevices`.

ERRORS, WARNINGS, AND NOTICES

Errors

`invalid_viewPort_reference`

SEE ALSO

To set a view port's view group, use the `GXSetViewPortViewGroup` function, described next.

The `gxScreenViewDevices` view group reference is described in the section "View Group Types" on page 7-69.

GXSetViewPortViewGroup

You can use the `GXSetViewPortViewGroup` function to assign a view port object to a new view group.

```
void GXSetViewPortViewGroup(gxViewPort target, gxViewGroup group);
```

`target` A reference to the view port whose view group you wish to change.

`group` A reference to the view group to which the view port is to be assigned.

DESCRIPTION

The `GXSetViewPortViewGroup` function assigns the target view port to the specified view group. Child view ports of the target view port are also assigned to that view group.

To assign a view port to the onscreen view group, pass the value `gxScreenViewDevices` for the `group` parameter. To obtain an offscreen view group reference to pass to this function, use the `GXNewViewGroup` function.

SPECIAL CONSIDERATIONS

The view ports in a hierarchy must all be in the same view group.

You cannot change the view group of a window view port (one attached to a Macintosh window).

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>invalid_viewPort_reference</code>	
<code>invalid_viewGroup_reference</code>	
<code>viewPort_is_a_window</code>	(debugging version)

Notices (debugging version)

<code>viewPort_already_in_viewGroup</code>
--

SEE ALSO

For an example of the use of this function, see Listing 7-2 on page 7-44.

To get a view port's view group, use the `GXGetViewPortViewGroup` function, described in the previous section.

The `gxScreenViewDevices` view group reference is described in the section "View Group Types" on page 7-69.

The `GXNewViewGroup` function is described on page 7-122.

GXGetViewPortAttributes

You can use the `GXGetViewPortAttributes` function to examine which attributes of a view port object are set.

```
gxPortAttribute GXGetViewPortAttributes(gxViewPort source);
```

`source` A reference to the view port whose attributes you wish to examine.

function result The view port attributes of the source view port.

ERRORS, WARNINGS, AND NOTICES**Errors**

`invalid_viewPort_reference`

SEE ALSO

View port attributes are described in the section “View Port Attributes” on page 7-20.

To set a view port’s attributes, use the `GXSetViewPortAttributes` function, described next.

GXSetViewPortAttributes

You can use the `GXSetViewPortAttributes` function to set or clear the attributes of a view port object.

```
void GXSetViewPortAttributes(gxViewPort target,
                             gxPortAttribute attributes);
```

`target` A reference to the view port whose attributes you wish to set.

`attributes` The new view port attributes to be assigned.

DESCRIPTION

The `GXSetViewPortAttributes` function sets the attributes of the view port object referenced in the `target` parameter to those specified in the `attributes` parameter. If you pass `gxNoAttributes` for the `attributes` parameter, all attributes are cleared.

ERRORS, WARNINGS, AND NOTICES**Errors**

`invalid_viewPort_reference`
`parameter_out_of_range` (debugging version)

Notices (debugging version)

`attributes_already_set`

SEE ALSO

For an example of the use of this function, see Listing 7-1 on page 7-42.

View port attributes are described in the section “View Port Attributes” on page 7-20.

To examine a view port’s attributes, use the `GXGetViewPortAttributes` function, described in the previous section.

GXGetViewPortTags

You can use the `GXGetViewPortTags` function to examine one or more of the tag objects associated with a view port object.

```
long GXGetViewPortTags(gxViewPort source, long tagType,
                      long index, long count, gxTag items[]);
```

<code>source</code>	A reference to the view port whose tag list you want to examine.
<code>tagType</code>	The type of tag object to search for. A value of 0 indicates that you want to look for all tag types.
<code>index</code>	The (1-based) index of the first such tag reference to return.
<code>count</code>	The number of tag references to return.
<code>items</code>	An array to hold the returned tag references.

function result The number of tag references found that fit the criteria.

DESCRIPTION

The `GXGetViewPortTags` function searches the tag list of the source view port object for references to tag objects with the tag type specified by the `tagType` parameter. If you specify 0 for the `tagType` parameter, the `GXGetViewPortTags` function searches all tag types.

You can use the `index` and the `count` parameters to specify which tag references of the appropriate type the `GXGetViewPortTags` function should return. The `index` parameter indicates the first tag reference to return and the `count` parameter indicates how many tag references to return. The `index` parameter must be greater than 0. The `count` parameter must be greater than 0 or equal to the `gxSelectToEnd` constant (-1), which indicates that all tag references (starting with the tag reference indicated by the `index` parameter) should be returned.

The function result is the number of tag references found that fit the criteria. If you pass a value other than `nil` for the `items` parameter, the `GXGetViewPortTags` function returns in it the tag references that were found.

Typically, you call this function once with a `nil` value for the `items` parameter to determine the number of matching tag references. Then you allocate an appropriately sized array and call the function a second time to obtain the references themselves.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>out_of_memory</code>	
<code>invalid_viewPort_reference</code>	
<code>index_is_less_than_one</code>	(debugging version)
<code>count_is_less_than_one</code>	(debugging version)

Warnings

<code>index_out_of_range</code>
<code>count_out_of_range</code>

SEE ALSO

Tag objects are discussed in the chapter “Tag Objects” in this book.

To change the set of tag references associated with a view port, use the `GXSetViewPortTags` function, described next.

GXSetViewPortTags

You can use the `GXSetViewPortTags` function to add, remove, or replace tag objects associated with a view port object.

```
void GXSetViewPortTags(gxViewPort target, long tagType,
                      long index, long oldCount,
                      long newCount, const gxTag items[]);
```

<code>target</code>	A reference to the view port whose tag list you want to alter.
<code>tagType</code>	The type of tag objects to replace. A value of 0 indicates that you want to replace tags of all types.
<code>index</code>	The (1-based) index of the first tag reference (to a tag object of the appropriate type) to replace.
<code>oldCount</code>	The number of tag references to replace. A value of 0 specifies that you want to insert tag references before the tag reference indicated by the <code>index</code> parameter, rather than replace tag references. A value of -1 (the <code>gxSelectToEnd</code> constant) specifies that all tag references of the requested type, starting with the tag reference indicated by the <code>index</code> parameter, should be replaced.
<code>newCount</code>	The number of tag references to insert. A value of 0 specifies that there are no tag references to insert; the existing tag references that match the criteria you specify are removed from the source shape’s tag list and disposed of.
<code>items</code>	An array of tag references to insert in the tag list.

DESCRIPTION

The `GXSetViewPortTags` function allows you add tag references to a view port object's tag list, to remove tag references from the list, or to replace tag references in the list with new tag references. In any of these three cases, the `target` parameter specifies the view port object to be modified, the `newCount` parameter specifies the number of tag references to add, and the `items` parameter provides the new tag references.

- n To add tag references, set the `oldCount` parameter to 0. Use the `tagType` and the `index` parameters to specify where to add the new tag references. (For example, if you specify `nil` for the `tagType` parameter and 1 for the `index` parameter, this function inserts the new tag references before the current tag references. If you specify a value other than `nil` for the `tagType` parameter and a value of 2 for the `index` parameter, the function inserts the new tag references before the second tag reference with a tag type matching the `tagType` parameter.)
- n To remove tag references, set the `newCount` parameter to 0 and the `items` parameter to `nil`. You can use the `index` and the `oldCount` parameters to specify which tag references (of the specified type) should be removed. The `index` parameter indicates the first tag reference (of the specified type) to remove and the `oldCount` parameter indicates how many tag references (of the specified type) to remove.
- n To replace tag references, use the `tagType`, `index`, and `oldCount` parameters to indicate which tag references to replace, and use the `newCount` and `items` parameters to specify the new tag references to add. If `newCount` is greater than `oldCount`, the extra tag references are placed immediately adjacent to the last tag reference replaced.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>out_of_memory</code>	
<code>invalid_viewPort_reference</code>	
<code>tag_is_nil</code>	
<code>parameter_is_nil</code>	(debugging version)
<code>inconsistent_parameters</code>	(debugging version)
<code>parameter_out_of_range</code>	(debugging version)
<code>index_is_less_than_zero</code>	(debugging version)
<code>cannot_dispose_locked_tag</code>	(debugging version)

Warnings

`index_out_of_range`
`count_out_of_range`

Notices (debugging version)

`tag_already_set`

SEE ALSO

Tag objects are discussed in the chapter “Tag Objects” in this book.

To examine the set of tag references associated with a view port, use the `GXGetViewPortTags` function, described in the previous section.

Retrieving the View Devices That Intersect a View Port

The function described in this section allows you to determine the view devices that a view port can draw to.

GXGetViewPortViewDevices

You can use the `GXGetViewPortViewDevices` function to determine all of the view device objects that shapes drawn to a view port can display on.

```
long GXGetViewPortViewDevices(gxViewPort source,
                              gxViewDevice list[]);
```

`source` A reference to the view port whose view devices you wish to examine.

`list` An array of view device references. On return, contains the references to the view devices that the source view port can draw to.

function result The number of view device references in the `list` array.

DESCRIPTION

The `GXGetViewPortViewDevices` function determines which view devices can display the contents of the source view port, and places a list of references to those view devices in the `list` parameter. It also returns the number of view devices in the list. The view devices returned are those, in the same view group as the view port, whose clip areas intersect the view port's clip area.

If you set the `list` parameter to `nil`, `GXGetViewPortViewDevices` does not return a list of references; it only returns the number of view device references that would be in the list. Thus, you typically call this function twice: first to get the size of array to allocate for the `list` parameter, and second to retrieve the list itself.

ERRORS, WARNINGS, AND NOTICES

Errors

`invalid_viewPort_reference`

SEE ALSO

For an example of the use of this function, see Listing 7-7 on page 7-49.

Retrieving the View Ports That Intersect a Shape

The function described in this section allows you to determine which view ports can display a shape object.

GXGetShapeGlobalViewPorts

You can use the `GXGetShapeGlobalViewPorts` function to determine all of the view ports that intersect the area of a shape.

```
long GXGetShapeGlobalViewPorts(gxShape source, gxViewPort list[]);
```

`source` A reference to the shape whose view ports you wish to examine.

`list` An array of view port references. On return, contains the references to the view ports that the shape can draw to.

function result The number of view port references in the `list` array.

DESCRIPTION

The `GXGetShapeGlobalViewPorts` function retrieves a list of the view ports that the source shape may draw to. If a view port is specified in the transform object associated with the shape, and if the transformed shape is not completely clipped by the view port's clip shape, the view port is put in the list returned by this function. The view ports need not all be in the same view group.

If you set the `list` parameter to `nil`, `GXGetShapeGlobalViewPorts` does not fill out the list of references; it only returns the number of view port references that would be in the list (which may be 0). Thus, you typically call this function twice: first to get the size of array to allocate for the `list` parameter, and second to retrieve the list itself.

As one application of this function, you could first call `GXGetShapeGlobalViewPorts` to determine the view ports to which a shape is drawn. You then could use the view ports in calls to `GXGetShapeGlobalViewDevices`, to determine the view devices a given shape would be drawn to.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`shape_is_nil`

SEE ALSO

The `GXGetShapeGlobalViewDevices` function is described on page 7-115.

Measuring a Shape in Local Coordinates

The function described in this section allows you to measure the size of a shape as it appears in its view ports—in local coordinates, after its transform mapping has been applied. Other QuickDraw GX functions are available to measure shapes in other contexts:

- n To determine a shape's bounding rectangle in global coordinates, use the `GXGetShapeGlobalBounds` function, described on page 7-125.
- n To determine a shape's bounding rectangle on a view device, use the `GXGetShapeDeviceBounds` function, described on page 7-116.
- n To determine a shape's bounding rectangle in geometry-space coordinates, use the `GXGetShapeBounds` function, described in the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*.

GXGetShapeLocalBounds

You can use the `GXGetShapeLocalBounds` function to determine the bounding rectangle of a shape in local coordinates.

```
gxRectangle *GXGetShapeLocalBounds(gxShape source,
                                   gxRectangle *bounds);
```

`source` A reference to the shape whose bounding rectangle you wish to determine in local coordinates.

`bounds` A pointer to a rectangle structure. On return, it contains the dimensions of the bounding rectangle.

function result A rectangle that defines the bounds of the shape in local coordinates. (It is the same as the rectangle returned in the `bounds` parameter.)

DESCRIPTION

The `GXGetShapeLocalBounds` function returns the bounding rectangle of the source shape after the shape's transform mapping and style have been applied. The dimensions of the rectangle are in the shape's local coordinates. The rectangle pointed to by the `bounds` parameter also receives the bounding rectangle in local coordinates.

To determine a shape's bounding rectangle in geometry-space coordinates, use the `GXGetShapeBounds` function. To determine a shape's bounding rectangle in global coordinates, use the `GXGetShapeGlobalBounds` function. To determine a shape's bounding rectangle on a view device, use the `GXGetShapeDeviceBounds` function.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`shape_is_nil`
`parameter_is_nil` (debugging version)

SEE ALSO

For an example of the use of this function, see Listing 7-8 on page 7-51.

The `GXGetShapeBounds` function is described in the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*. The `GXGetShapeGlobalBounds` function is described on page 7-125. The `GXGetShapeDeviceBounds` function is described on page 7-116.

For information about coordinate spaces, see the section "About Drawing, Coordinate Conversion, and Clipping" beginning on page 7-30.

View Device Functions

This section describes the QuickDraw GX functions you use with view device objects. Using the functions described here, you can

- n create and manipulate view device objects and their properties
- n retrieve the view devices that intersect a shape
- n measure and analyze a shape on a device, including hit-testing a shape on a device

Creating and Manipulating View Device Objects

The functions described in this section allow you to create and manipulate view device objects. With the functions in this section, you can

- n create and dispose of view devices
- n copy view devices
- n test view devices for equality

GXNewViewDevice

You can use the `GXNewViewDevice` function to create a new view device object.

```
gxViewDevice GXNewViewDevice (gxViewGroup group,
                               gxShape bitmapShape);
```

`group` A reference to the view group in which to create the view device.

`bitmapShape` A reference to a bitmap shape that defines the view device's imaging area.

function result A reference to the newly created view device object.

DESCRIPTION

The `GXNewViewDevice` function creates a new view device object in the specified view group. The `bitmapShape` parameter references a bitmap shape whose bitmap structure specifies the height, width, and pixel depth (bits per pixel) of the device, plus any color set or color profile used by the device. The remaining properties have default values:

- n a clip that is a full shape
- n a mapping that is the identity mapping
- n no attributes set
- n an empty tag list

To obtain an offscreen view group reference to pass to this function, use the `GXNewViewGroup` function. To create a view device in the onscreen view group, pass the value `gxScreenViewDevices` for the `group` parameter.

SPECIAL CONSIDERATIONS

The bitmap shape that you pass to this function cannot not have a disk-based pixel image.

If no error occurs, the `GXNewViewDevice` function creates a view device object; you are responsible for disposing of that object when you no longer need it.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
shape_is_nil
invalid_viewGroup_reference
illegal_type_for_shape           (debugging version)
```


SEE ALSO

For examples of the use of this function, see Listing 7-9 on page 7-53 and Listing 7-14 on page 7-63.

To dispose of a view device, use the `GXDisposeViewDevice` function, described next. To dispose of all the view devices in a view group, use the `GXDisposeViewGroup` function, described on page 7-122.

For information about bitmap shapes and the bitmap structure, see the bitmap shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*.

The `GXNewViewGroup` function is described on page 7-122. The `gxScreenViewDevices` view group reference is described in the section “View Group Types” on page 7-69.

GXDisposeViewDevice

You can use the `GXDisposeViewDevice` function to delete a view device object.

```
void GXDisposeViewDevice(gxViewDevice target);
```

target A reference to the view device.

DESCRIPTION

The `GXDisposeViewDevice` function disposes of the view device and all associated memory structures, including the view device’s clip shape, bitmap, color set, and color profile.

ERRORS, WARNINGS, AND NOTICES**Errors**

`invalid_viewDevice_reference`

SEE ALSO

For an example of the use of this function, see page 7-53.

To dispose of all the view devices in a view group, use the `GXDisposeViewGroup` function, described on page 7-122.

GXCopyToViewDevice

You can use the `GXCopyToViewDevice` function to create a copy of an existing view device object.

```
gxViewDevice GXCopyToViewDevice (gxViewDevice target,
                                  gxViewDevice source);
```

target A reference to the view device to copy the source contents into. If you specify `nil` for this parameter, the `GXCopyToViewDevice` function creates a new view device object.

source A reference to the view device whose contents you want to copy.

function result A reference to the view device that is a copy of the source view device.

DESCRIPTION

The `GXCopyToViewDevice` function copies the contents of an existing view device object to another, or it creates a new view device object and copies the contents of an existing view device object to it. The function copies the clip, mapping, bitmap shape (including color space, color profile reference, and color set reference), attributes, and tag list from the source view device. It clones, but does not copy, the tag objects in the tag list.

In copying the bitmap, the `GXCopyToViewDevice` function does not copy the pixel image itself, just the properties of the bitmap shape. For the color set and color profile properties of the bitmap, and tag objects, `GXCopyToViewDevice` copies only the references, not the objects themselves.

If you specify `nil` for the `target` parameter, the `GXCopyToViewDevice` function creates a new view port object and copies the properties of the source view device into it.

SPECIAL CONSIDERATIONS

If you attempt to copy to a screen device, this function posts a `viewDevice_access_restricted` error.

If you specify `nil` for the `target` parameter and no error occurs, the `GXCopyToViewDevice` function creates a view device object; you are responsible for disposing of that object when you no longer need it.

ERRORS, WARNINGS, AND NOTICES

Errors

<code>out_of_memory</code>	
<code>invalid_viewDevice_reference</code>	
<code>viewDevice_access_restricted</code>	(debugging version)

SEE ALSO

To create a new view device that has default properties instead of being a copy of an existing view device, use the `GXNewViewDevice` function, described on page 7-98.

To compare two view device objects for equality, use the `GXEqualViewDevice` function, described next.

GXEqualViewDevice

You can use the `GXEqualViewDevice` function to determine whether two view device objects are equal.

```
boolean GXEqualViewDevice(gxViewDevice one, gxViewDevice two);
```

`one` A reference to one view device to test for equality.

`two` A reference to another view device to test for equality.

function result `true` if the view device specified by the `one` parameter is equal to the view device specified by the `two` parameter; `false` otherwise.

DESCRIPTION

The `GXEqualViewDevice` function returns as its function result a Boolean value indicating whether the view device object specified by the `one` parameter is equal to the view device object specified by the `two` parameter.

For two view device objects to be equal, they must have identical clips, mappings, bitmap shapes, and attributes. They also must be in the same view group, represent the same Macintosh graphics device (same `GDevice` record), and point to the same pixel image, color set, and color profile. The view device objects are not equal if they point to different but equivalent pixel images, color sets, or color profiles. The tag lists of the view devices need not be identical.

ERRORS, WARNINGS, AND NOTICES**Errors**

`invalid_viewDevice_reference`

SEE ALSO

To make a copy of a view device object that is equal by the criteria of this function, use the `GXCopyToViewDevice` function, described in the previous section.

Macintosh graphics devices and the `GDevice` record are described in *Inside Macintosh: Imaging With QuickDraw*. The relationship of view devices to `GDevice` records is discussed in the Macintosh environment chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

Manipulating View Device Object Properties

The functions described in this section allow you to create and manipulate view device objects. With these functions, you can get and set a view device's

- n clip
- n mapping
- n bitmap
- n view group
- n attributes
- n tag list

GXGetViewDeviceClip

You can use the `GXGetViewDeviceClip` function to examine the clip property of a view device object.

```
gxShape GXGetViewDeviceClip(gxViewDevice source);
```

`source` A reference to the view device whose clip you wish to examine.

function result A reference to a shape object whose geometry defines the view device's clip.

DESCRIPTION

The `GXGetViewDeviceClip` function returns a shape object that defines the geometry of the clip associated with the view device. The function returns `nil` if there is no clip. The clip shape is a copy of the view device's clip; changing this shape does not change the view device's clip.

SPECIAL CONSIDERATIONS

If no error occurs, the `GXGetViewDeviceClip` function creates a shape object; you are responsible for disposing of that object when you no longer need it.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`invalid_viewDevice_reference`

SEE ALSO

To set a view device's clip, use the `GXSetViewDeviceClip` function, described next.

GXSetViewDeviceClip

You can use the `GXSetViewDeviceClip` function to set the clip property of a view device object.

```
void GXSetViewDeviceClip(gxViewDevice target, gxShape clip);
```

<code>target</code>	A reference to the view device whose clip you wish to set.
<code>clip</code>	A reference to a shape object whose geometry describes the clip to be assigned.

DESCRIPTION

The `GXSetViewDeviceClip` function copies information from the shape object referenced by the `clip` parameter into the clip property of the view device object referenced by the `target` parameter. You can specify `nil` for the `clip` parameter, in which case this function sets the clip property of the target view device to a full clip. (A full clip indicates that QuickDraw GX is not to apply view device clipping to shapes that reference this view device.)

Although a filled rectangle is the most common clip shape for a view device, the new clip shape may be a geometric shape, a bitmap shape, or a glyph shape. It may not be a picture, text, or layout shape.

- n If you specify a geometric shape, it must be in primitive form—that is, all the stylistic information about the shape must be incorporated into the shape's geometry—because this function copies only the geometry-related information from the shape you specify. It does not copy the information contained in the shape's style. You can convert a shape to its primitive form using the `GXPrimitiveShape` function, which is described in *Inside Macintosh: QuickDraw GX Graphics*. You can also specify an empty or full shape for a clip.

View-Related Objects

- n If you specify a bitmap shape, it must have a pixel size of 1 and its color profile reference must be `nil`. In the bitmap, pixel values of 0 obscure drawing; pixel values of 1 do not restrict visibility. The `GXSetViewDeviceClip` function copies the pixel image from the bitmap to the clip property of the target view device.
- n If you specify a glyph shape, this function uses information from the glyph shape's style object as well as its style list to determine the size, form, and position of the glyph outlines; those outlines are then used to clip drawing. The style list cannot have `nil` entries. A style object referenced by the glyph shape cannot be complex—that is, it cannot have a cap, join, dash, pattern, text face, font variation, tag list, or any of the properties used only by layout shapes.

You only need to call this function if you want to restrict the part of the device that displays your view ports—for example, to clip out the area reserved for live video in your offscreen copy of an onscreen view device.

Because it is copied into the view device object, changing the clip shape after calling `GXSetViewDeviceClip` does not affect the view device's clip.

SPECIAL CONSIDERATIONS

You cannot change the clip of a view device in the onscreen view group.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>out_of_memory</code>	
<code>invalid_viewDevice_reference</code>	
<code>colorProfile_must_be_nil</code>	(debugging version)
<code>bitmap_pixel_size_must_be_1</code>	(debugging version)
<code>empty_shape_not_allowed</code>	(debugging version)
<code>ignorePlatformShape_not_allowed</code>	(debugging version)
<code>nil_style_in_glyph_not_allowed</code>	(debugging version)
<code>complex_glyph_style_not_allowed</code>	(debugging version)
<code>illegal_type_for_shape</code>	(debugging version)
<code>shapeFill_not_allowed</code>	(debugging version)
<code>viewDevice_access_restricted</code>	(debugging version)

Notices (debugging version)

`clip_already_set`
`tags_in_shape_ignored`

SEE ALSO

For information about geometric shapes and bitmap shapes, see *Inside Macintosh: QuickDraw GX Graphics*. For information about glyph shapes, see *Inside Macintosh: QuickDraw GX Typography*.

To retrieve a copy of the clip property, use the `GXGetViewDeviceClip` function, described in the previous section.

GXGetViewDeviceMapping

You can use the `GXGetViewDeviceMapping` function to examine the mapping property of a view device object.

```
gxMapping *GXGetViewDeviceMapping(gxViewDevice source,
                                   gxMapping *map);
```

`source` A reference to the view device whose mapping you wish to examine.

`map` A pointer to a mapping structure. On return, the structure contains a copy of the mapping matrix of the source view device.

function result A pointer to the mapping matrix of the source view device. (This value is the same as the value returned in the `map` parameter.)

DESCRIPTION

The `GXGetViewDeviceMapping` function copies the mapping matrix information from the mapping property of the source view device object into the mapping structure pointed to by the `map` parameter. The function also returns as its function result a pointer to this mapping structure.

To make changes to the source view device's mapping property, you can alter the information returned by this function, and then use the `GXSetViewDeviceMapping` function to reassign the altered mapping to the source view device.

ERRORS, WARNINGS, AND NOTICES

Errors

```
invalid_viewDevice_reference
parameter_is_nil                    (debugging version)
```

SEE ALSO

For an example of the use of this function, see Listing 7-11 on page 7-57.

To set a view device's mapping, use the `GXSetViewDeviceMapping` function, described next.

For information about the `gxMapping` structure, see the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

GXSetViewDeviceMapping

You can use the `GXSetViewDeviceMapping` function to assign a mapping to a view device object.

```
void GXSetViewDeviceMapping(gxViewDevice target,
                           const gxMapping *map);
```

`target` A reference to the view device object you want to assign the mapping to.
`map` A pointer to a mapping structure containing the mapping matrix to assign to the target view device.

DESCRIPTION

The `GXSetViewDeviceMapping` function copies the mapping structure pointed to by the `map` parameter into the mapping property of the target view device.

You can specify `nil` for the `map` parameter. If you do, the mapping is set to a translation that prevents the view device from overlapping any other view device in the view device's view group. This translation may cause the view device to move to an area adjacent to, but not overlapping, other view devices in this view group.

You can provide arbitrary values for the elements of the mapping structure pointed to by the `map` parameter, with one exception: the lower-right element of this matrix (element [2][2]) may not be 0.

SPECIAL CONSIDERATIONS

You cannot change the mapping of a view device in the onscreen view group.

ERRORS, WARNINGS, AND NOTICES

Errors

`invalid_viewDevice_reference`
`viewDevice_access_restricted` (debugging version)

SEE ALSO

For an example of the use of this function, see page 7-57.

To retrieve a view device's mapping, use the `GXGetViewDeviceMapping` function, described in the previous section.

For information about the `gxMapping` structure, see the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

GXGetViewDeviceBitmap

You can use the `GXGetViewDeviceBitmap` function to retrieve the bitmap property of a view device object.

```
gxShape GXGetViewDeviceBitmap(gxViewDevice source);
```

source A reference to the view device whose bitmap you wish to examine.

function result A bitmap shape object that is a copy of the information in the view device's bitmap property.

DESCRIPTION

The `GXGetViewDeviceBitmap` function returns a bitmap shape that is a copy of the bitmap representing the imaging area of the specified device. The pixel image pointer in the bitmap structure of the bitmap shape may be `nil` if, for example, the image is on disk. The pointer is not `nil` if the image is associated with an onscreen device or was supplied by an application.

The shape returned by this function is a copy of the device's bitmap, so if you alter it you must reassign it to the view device with the `GXSetViewDeviceBitmap` function. However, the pixel image of the bitmap shape returned by this function is *not* a copy; it is the same pixel image as that used by the view device. Thus if you draw to the view device you modify the pixel image of the returned shape, and likewise if you modify the pixel image of the returned shape you modify the pixel image of the view device. You can take advantage of this fact to draw an offscreen bitmap directly to an onscreen view device.

SPECIAL CONSIDERATIONS

If no error occurs, the `GXGetViewDeviceBitmap` function creates a shape object; you are responsible for disposing of that object when you no longer need it.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`invalid_viewDevice_reference`

SEE ALSO

For examples of the use of this function, see page 7-55 and Listing 7-14 on page 7-63.

To set a view devices's bitmap property, use the `GXSetViewDeviceBitmap` function, described next.

For information about bitmap shapes, the bitmap structure, and pixel images, see the bitmap shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*.

GXSetViewDeviceBitmap

You can use the `GXSetViewDeviceBitmap` function to set the bitmap property of a view device object.

```
void GXSetViewDeviceBitmap(gxViewDevice target,
                           gxShape bitmapShape);
```

`target` A reference to the view device whose bitmap you wish to set.

`bitmapShape` A reference to a bitmap shape object that specifies the new bitmap.

DESCRIPTION

The `GXSetViewDeviceBitmap` function sets the bitmap property in the view device to contain the information in the shape specified in the `bitmapShape` parameter. Only the bitmap structure in the geometry of the bitmap shape is copied into the view device.

SPECIAL CONSIDERATIONS

The bitmap shape you supply to this function cannot have a disk-based pixel image.

ERRORS, WARNINGS, AND NOTICES

Errors

<code>out_of_memory</code>	
<code>invalid_viewDevice_reference</code>	
<code>parameter_is_nil</code>	(debugging version)
<code>illegal_type_for_shape</code>	(debugging version)
<code>viewDevice_access_restricted</code>	(debugging version)

SEE ALSO

For an example of the use of this function, see page 7-55.

To retrieve the bitmap property of a view device, use the `GXGetViewDeviceBitmap` function, described in the previous section.

For information about bitmap shapes, see the bitmap shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*.

GXGetViewDeviceViewGroup

You can use the `GXGetViewDeviceViewGroup` function to determine the view group that a view device is part of.

```
gxViewGroup GXGetViewDeviceViewGroup(gxViewDevice source);
```

`source` A reference to the view device whose view group you wish to examine.

function result A reference to the view group that the source view device is part of.

DESCRIPTION

The `GXGetViewDeviceViewGroup` function returns a reference to the source view device's view group. If it is the onscreen view group, the returned value is `gxScreenViewDevices`.

ERRORS, WARNINGS, AND NOTICES

Errors

`invalid_viewDevice_reference`

SEE ALSO

To set a view device's view group, use the `GXSetViewDeviceViewGroup` function, described next.

The `gxScreenViewDevices` view group reference is described in the section "View Group Types" on page 7-69.

GXSetViewDeviceViewGroup

You can use the `GXSetViewDeviceViewGroup` function to assign a view device object to a specified view group.

```
void GXSetViewDeviceViewGroup(gxViewDevice target,
                               gxViewGroup group);
```

`target` A reference to the view device whose view group you wish to change.

`group` A reference to the view group to which the view device is to be assigned.

DESCRIPTION

The `GXSetViewDeviceViewGroup` function changes the target view device to the specified view group.

SPECIAL CONSIDERATIONS

You cannot assign a view device to the onscreen view group; do not specify `gxScreenViewDevices` for the `group` parameter. Also, you cannot change the view group of a view device already in the onscreen view group.

ERRORS, WARNINGS, AND NOTICES**Errors**

`invalid_viewDevice_reference`
`invalid_viewGroup_reference`
`viewDevice_access_restricted` (debugging version)

Notices (debugging version)

`viewDevice_already_in_viewGroup`

SEE ALSO

For an example of the use of this function, see Listing 7-10 on page 7-54.

To get a view device's view group, use the `GXGetViewDeviceViewGroup` function, described in the previous section.

The `gxScreenViewDevices` view group reference is described in the section "View Group Types" on page 7-69.

GXGetViewDeviceAttributes

You can use the `GXGetViewDeviceAttributes` function to determine which attributes of a view device object are set.

```
gxDeviceAttribute GXGetViewDeviceAttributes(gxViewDevice source);
```

`source` A reference to the view device whose attributes you wish to examine.

function result The view device attributes of the source view device.

ERRORS, WARNINGS, AND NOTICES**Errors**

`invalid_viewDevice_reference`

SEE ALSO

View device attributes are described in the section “View Device Attributes” on page 7-27.

To set a view device’s attributes, use the `GXSetViewDeviceAttributes` function, described next.

GXSetViewDeviceAttributes

You can use the `GXSetViewDeviceAttributes` function to set or clear the attributes of a view device object.

```
void GXSetViewDeviceAttributes(gxViewDevice target,
                              gxDeviceAttribute attributes);
```

`target` A reference to the view device whose attributes you wish to set.

`attributes` A reference to the view port’s attributes.

DESCRIPTION

The `GXSetViewDeviceAttributes` function sets the attributes of the view device object referenced in the `target` parameter to those specified in the `attributes` parameter. If you pass `gxNoAttributes` for the `attributes` parameter, all attributes are cleared.

You cannot change the attributes of a view device in the onscreen view group.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>invalid_viewDevice_reference</code>	
<code>parameter_out_of_range</code>	(debugging version)
<code>viewDevice_access_restricted</code>	(debugging version)

Notices (debugging version)

<code>attributes_already_set</code>

SEE ALSO

View device attributes are described in the section “View Device Attributes” on page 7-27.

To get a view device’s attributes, use the `GXGetViewDeviceAttributes` function, described in the previous section.

GXGetViewDeviceTags

You can use the `GXGetViewDeviceTags` function to examine one or more of the tag objects associated with a view device object.

```
long GXGetViewDeviceTags(gxViewDevice source, long tagType,
                        long index, long count, gxTag items[]);
```

<code>source</code>	A reference to the view device object whose tag list you want to examine.
<code>tagType</code>	The type of tag object to search for. A value of 0 indicates that you want to look for all tag types.
<code>index</code>	The (1-based) index of the first such tag reference to return.
<code>count</code>	The number of tag references to return.
<code>items</code>	An array to hold the returned tag references.

function result The number of tag references found that fit the criteria.

DESCRIPTION

The `GXGetViewDeviceTags` function searches the tag list of the source view device object for references to tag objects with the tag type specified by the `tagType` parameter. If you specify 0 for the `tagType` parameter, the `GXGetViewDeviceTags` function searches all tag types.

You can use the `index` and the `count` parameters to specify which tag references of the appropriate type the `GXGetViewDeviceTags` function should return. The `index` parameter indicates the first tag reference to return and the `count` parameter indicates how many tag references to return. The `index` parameter must be greater than 0. The `count` parameter must be greater than 0 or equal to the `gxSelectToEnd` constant (-1), which indicates that all tag references (starting with the tag reference indicated by the `index` parameter) should be returned.

The function result is the number of tag references found that fit the criteria. If you pass a value other than `nil` for the `items` parameter, the `GXGetViewDeviceTags` function returns in it the tag references that were found.

Typically, you call this function once with a `nil` value for the `items` parameter to determine the number of matching tag references. Then you allocate an appropriately sized array and call the function a second time to obtain the references themselves.

ERRORS, WARNINGS, AND NOTICES**Errors**

```

out_of_memory
invalid_viewDevice_reference
index_is_less_than_one           (debugging version)
count_is_less_than_one          (debugging version)

```

Warnings

```

index_out_of_range
count_out_of_range

```

SEE ALSO

Tag objects are discussed in the chapter “Tag Objects” in this book.

To change the set of tag references associated with a view device, use the `GXSetViewDeviceTags` function, described next.

GXSetViewDeviceTags

You can use the `GXSetViewDeviceTags` function to add, remove, or replace tag objects associated with a view device object.

```

void GXSetViewDeviceTags(gxViewDevice target, long tagType,
                        long index, long oldCount, long newCount,
                        const gxTag items[]);

```

<code>target</code>	A reference to the view device object whose tag list you want to alter.
<code>tagType</code>	The type of tag objects to replace. A value of 0 indicates that you want to replace tags of all types.
<code>index</code>	The (1-based) index of the first tag reference (to a tag object of the appropriate type) to replace.
<code>oldCount</code>	The number of tag references to replace. A value of 0 specifies that you want to insert tag references before the tag reference indicated by the <code>index</code> parameter, rather than replace tag references. A value of -1 (the <code>gxSelectToEnd</code> constant) specifies that all tag references of the requested type, starting with the tag reference indicated by the <code>index</code> parameter, should be replaced.
<code>newCount</code>	The number of tag references to insert. A value of 0 specifies that there are no tag references to insert; the existing tag references that match the criteria you specify are removed from the source shape’s tag list and disposed of.
<code>items</code>	An array of tag references to insert in the tag list.

DESCRIPTION

The `GXSetViewDeviceTags` function allows you add tag references to a view device object's tag list, to remove tag references from the list, or to replace tag references in the list with new tag references. In any of these three cases, the `target` parameter specifies the view port device to be modified, the `newCount` parameter specifies the number of tag references to add, and the `items` parameter provides the new tag references.

- n To add tag references, set the `oldCount` parameter to 0. Use the `tagType` and the `index` parameters to specify where to add the new tag references. (For example, if you specify `nil` for the `tagType` parameter and 1 for the `index` parameter, this function inserts the new tag references before the current tag references. If you specify a value other than `nil` for the `tagType` parameter and a value of 2 for the `index` parameter, the function inserts the new tag references before the second tag reference with a tag type matching the `tagType` parameter.)
- n To remove tag references, set the `newCount` parameter to 0 and the `items` parameter to `nil`. You can use the `index` and the `oldCount` parameters to specify which tag references (of the specified type) should be removed. The `index` parameter indicates the first tag reference (of the specified type) to remove and the `oldCount` parameter indicates how many tag references (of the specified type) to remove.
- n To replace tag references, use the `tagType`, `index`, and `oldCount` parameters to indicate which tag references to replace, and use the `newCount` and `items` parameters to specify the new tag references to add. If `newCount` is greater than `oldCount`, the extra tag references are placed immediately adjacent to the last tag reference replaced.

You cannot change the tag list of a view device in the onscreen view group.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>out_of_memory</code>	
<code>invalid_viewDevice_reference</code>	
<code>tag_is_nil</code>	
<code>parameter_is_nil</code>	(debugging version)
<code>inconsistent_parameters</code>	(debugging version)
<code>parameter_out_of_range</code>	(debugging version)
<code>index_is_less_than_zero</code>	(debugging version)
<code>cannot_dispose_locked_tag</code>	(debugging version)

Warnings

`index_out_of_range`
`count_out_of_range`

Notices (debugging version)

`tag_already_set`

SEE ALSO

Tag objects are discussed in the chapter “Tag Objects” in this book.

To examine the set of tag references associated with a view port, use the `GXGetViewDeviceTags` function, described in the previous section.

Retrieving the View Devices That Intersect a Shape

The function described in this section allows you to get the view devices on which a shape appears.

GXGetShapeGlobalViewDevices

You can use the `GXGetShapeGlobalViewDevices` function to determine which view devices intersect the area of a shape.

```
long GXGetShapeGlobalViewDevices(gxShape source, gxViewPort port,
                                gxViewDevice list[]);
```

<code>source</code>	A reference to the shape whose view devices you wish to retrieve.
<code>port</code>	A reference to a view port that the shape is drawn to.
<code>list</code>	An array of view device references. On return, the array lists the view devices that the shape is drawn on.

function result The number of view devices that the shape is drawn on.

DESCRIPTION

The `GXGetShapeGlobalViewDevices` function retrieves a list of view devices that the source shape is drawn on, through the specified view port. The view port must be in the view port list of the shape’s transform object. A returned view device object must have a clip that intersects the clip of the specified view port, and the shape drawn to the view port must also intersect the view device clip.

If the `port` parameter is set to `nil`, all view ports in the view port list of the shape’s transform object are used.

If you set the `list` parameter to `nil`, `GXGetShapeGlobalViewDevices` does not fill out the list of references; it only returns the number of view device references that would be in the list (which may be 0). Thus, you typically call this function twice: first to get the size of array to allocate for the `list` parameter, and second to retrieve the list itself.

As one application of this function, you could call `GXGetShapeGlobalViewPorts` to determine the view ports to which a shape is drawn. You then could use one of these view ports in a call to `GXGetShapeGlobalViewDevices` to determine the view devices that the shape would be drawn to.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`shape_is_nil`
`invalid_viewPort_reference`

SEE ALSO

For examples of the use of this function, see Listing 7-12 on page 7-58 and Listing 7-13 on page 7-61.

The `GXGetShapeGlobalViewPorts` function is described on page 7-95.

Measuring a Shape in Device Coordinates

The functions described in this section allow you to get a shape's bounding rectangle and its area, as measured on a view device. Other QuickDraw GX functions are available to measure shapes in other contexts:

- n To determine a shape's bounding rectangle in local coordinates, use the `GXGetShapeLocalBounds` function, described on page 7-96.
- n To determine a shape's bounding rectangle in global coordinates, use the `GXGetShapeGlobalBounds` function, described on page 7-125.
- n To determine a shape's bounding rectangle in geometry-space coordinates, use the `GXGetShapeBounds` function, described in the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*.

GXGetShapeDeviceBounds

You can use the `GXGetShapeDeviceBounds` function to determine the bounding rectangle for the visible part of a shape on a view device.

```
boolean GXGetShapeDeviceBounds(gxShape source, gxViewPort port,
                               gxViewDevice device,
                               gxRectangle *bounds);
```

`source` A reference to the shape whose bounding rectangle you wish to test for inclusion on a device.

`port` A reference to a view port to which the shape is drawn.

`device` A reference to the view device that the shape displays on.

`bounds` A pointer to a rectangle structure. On return the structure contains the bounding rectangle, in device coordinates, for the part of the shape that appears on the device.

function result `true` if the bounding rectangle overlaps the view device clip; `false` if it does not.

DESCRIPTION

The `GXGetShapeDeviceBounds` function returns a value specifying whether the bounding rectangle of the shape drawn to the specified view port is within the clip area of the specified view device. The view port and view device must be in the same view group. The view port must be in the view port list of the shape's transform object.

You can specify `nil` for the `port` parameter, in which case `GXGetShapeDeviceBounds` includes all view ports in the view port list of the source shape's transform. You can specify `nil` for the `device` parameter, in which case `GXGetShapeDeviceBounds` includes any view device that intersects any of the specified view ports.

Unless you pass `nil` for the `bounds` parameter, the function also returns in the `bounds` parameter the part of the shape's bounding rectangle that displays on the device. The rectangle shape is defined in device coordinates.

To determine a shape's bounding rectangle in geometry-space coordinates, use the `GXGetShapeBounds` function. To determine a shape's bounding rectangle in local coordinates, use the `GXGetShapeLocalBounds` function. To determine a shape's bounding rectangle in global coordinates, use the `GXGetShapeGlobalBounds` function.

SPECIAL CONSIDERATIONS

If the bounding rectangle of the source shape spans more than one device, this function posts an `unable_to_get_bounds_on_multiple_devices` warning.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`shape_is_nil`
`invalid_viewPort_reference`
`invalid_viewDevice_reference`
`inconsistent_parameters` (debugging version)

Warnings

`unable_to_get_bounds_on_multiple_devices`

SEE ALSO

For an example of the use of this function, see page 7-59.

To calculate the area of a shape on a device, use the `GXGetShapeDeviceArea` function, described next.

The `GXGetShapeBounds` function is described in the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*. The `GXGetShapeLocalBounds` function is described on page 7-96. The `GXGetShapeGlobalBounds` function is described on page 7-125.

For information about coordinate spaces, see the section "About Drawing, Coordinate Conversion, and Clipping" beginning on page 7-30.

GXGetShapeDeviceArea

You can use the `GXGetShapeDeviceArea` function to calculate the area of a shape on a given view device.

```
long GXGetShapeDeviceArea(gxShape source, gxViewPort port,
                          gxViewDevice device);
```

`source` A reference to the shape whose area on the device you want to determine.
`port` A reference to a view port that the shape is drawn to.
`device` A reference to the view device for which you wish to calculate the area.

function result The number of pixels that represent the shape on the device.

DESCRIPTION

The `GXGetShapeDeviceArea` function returns the number of pixels covered by the source shape in the view port on the specified view device. The shape object cannot be a bitmap or picture shape. The view port must be in the view port list of the shape's transform object.

This function accounts for just the pixels that would actually be affected if the shape were drawn. Spaces inside of framed or filled geometric shapes, or spaces within and between glyphs of typographic shapes, do not contribute to the calculated area.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
shape_is_nil
invalid_viewPort_reference
invalid_viewDevice_reference
illegal_type_for_shape                    (debugging version)
```

SEE ALSO

To determine the bounding rectangle of a shape on a device, use the `GXGetShapeDeviceBounds` function, described in the previous section.

To determine the area of a shape in geometry-space coordinates, use the `GXGetShapeArea` function, described in the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*.

Measuring the Colors and Pattern Width of a Shape on a Device

The function described in this section allows you to determine the exact colors and size of pattern that QuickDraw GX will use to draw a shape on a given view device.

GXGetShapeDeviceColors

You can use the `GXGetShapeDeviceColors` function to determine the set of colors with which a shape will be drawn on a given view device, as well as the width of any repeating pattern with which the shape will be drawn.

```
gxColorSet GXGetShapeDeviceColors(gxShape source,
                                   gxViewPort port,
                                   gxViewDevice device,
                                   long *width);
```

<code>source</code>	A reference to the shape whose colors you wish to determine.
<code>port</code>	A reference to a view port to which the shape is drawn.
<code>device</code>	A reference to the view device on which the shape is drawn.
<code>width</code>	A pointer to a <code>long</code> value. On return, the value is the width of the repeated pattern formed by dithering or halftoning, or by the pattern—if any—specified in the shape's style object.

function result A reference to a color set that contains the colors with which the shape can be drawn.

DESCRIPTION

The `GXGetShapeDeviceColors` function returns a color set containing the colors that the shape would be drawn with through the view port onto the view device. The view port must be in the view port list of the shape's transform object. If no shape sharing the ink of the source shape intersects the view port and view device, the function returns `nil`.

The `GXGetShapeDeviceColors` function returns only the number of unique colors in the dither pattern or the halftone pattern, not the size of the dither or the halftone. It also does not take transfer modes into account, or the colors already on the view device.

This function does not check that the shape actually intersects the view device; you may want to call the `GXGetShapeGlobalViewDevices` function first. The shape object cannot be a bitmap or picture shape.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
shape_is_nil
invalid_viewPort_reference
invalid_viewDevice_reference
```

SEE ALSO

For information about color sets, see “When Color Matching Occurs” beginning on page 4-31.

For information about dithers and halftones, see the sections “Dither” beginning on page 7-10 and “Halftone” beginning on page 7-13.

Patterns and the style object are described in the geometric styles chapter of *Inside Macintosh: QuickDraw GX Graphics*.

To determine the view devices that a shape is displayed on, use the `GXGetShapeGlobalViewDevices` function, described on page 7-115.

Hit-Testing a Shape on a Device

The function described in this section allows you to hit-test a shape in relation to the pixels of a view device.

GXHitTestDevice

You can use the `GXHitTestDevice` function to determine whether a point in device space is within a given tolerance of a shape displayed on that device.

```
gxShape GXHitTestDevice(gxShape target, gxViewPort port,
                        gxViewDevice device, const gxPoint *test,
                        const gxPoint *tolerance);
```

<code>target</code>	A reference to the shape to hit-test.
<code>port</code>	A reference to a view port that the shape is drawn to.
<code>device</code>	A reference to the view device on which the shape is drawn.
<code>test</code>	A pointer to a point structure specifying the location, in device coordinates (pixels), to hit-test the shape against.
<code>tolerance</code>	A pointer to a point structure specifying a rectangular shape whose size specifies the distance, in pixels, from the target shape that the test point can be and still be considered a successful hit.

function result A reference to the target shape if the shape was hit; otherwise `nil`.

DESCRIPTION

The `GXHitTestDevice` function returns the target shape within the specified view port if the hit is successful, otherwise it returns `nil`. All clipping, from transform through view port and view device, is taken into account in determining whether a hit is possible.

The test point represents a pixel location in view device coordinates. The tolerance represents a rectangular area of pixels, defining the “radius” of the total tolerance area.

You can think of four such rectangles as making up a larger rectangle centered on the hit point; if the distance from the hit point to the shape is within that larger rectangle, the hit is considered successful.

Negative values for the tolerance are permitted.

If the `port` parameter is set to 0, all view ports on the view device are tested. If the `device` parameter is set to 0, all view devices intersected by the view port are tested. If both `port` and `device` parameters are set to 0, all view ports that the shape is drawn to and all view devices drawn to by the target shape are tested.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`shape_is_nil`
`parameter_is_nil` (debugging version)

SEE ALSO

To hit-test individual parts of a shape's geometry, use the `GXHitTestShape` function, described in the chapter "Shape Objects" in this book. To hit-test the parts of a picture shape, use the `GXHitTestPicture` function, described in the picture shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*. To hit-test the text of a layout shape, use the `GXHitTestLayout` function, described in the layout carets chapter of *Inside Macintosh: QuickDraw GX Typography*.

For more information on `GXHitTestDevice` and how it relates to the other hit-testing functions, see "Hit-Testing a Shape on a Device" on page 7-60.

View Group Functions

This section describes the QuickDraw GX functions you use with view group objects. Using the functions described here, you can

- n create and dispose of view group objects
- n determine the view ports and view devices that belong to a view group
- n measure a shape in a view group's global space

Creating and Disposing of View Group Objects

The functions described in this section allow you to create and dispose of view groups.

GXNewViewGroup

You can use the `GXNewViewGroup` function to create a new view group object.

```
gxViewGroup GXNewViewGroup(void);
```

function result A reference to the new view group.

DESCRIPTION

The `GXNewViewGroup` function returns a unique view group reference. You can then use the new view group to create view ports and view devices that share the same global space. QuickDraw GX provides an onscreen view group, `gxScreenViewDevices`, for you. You only need to create offscreen view groups.

SPECIAL CONSIDERATIONS

If no error occurs, the `GXNewViewGroup` function creates a view group object; you are responsible for disposing of that object when you no longer need it.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`

SEE ALSO

For examples of the use of this function, see Listing 7-13 on page 7-61 and Listing 7-14 on page 7-63.

For information about view groups, see “About View Group Objects” beginning on page 7-29.

To dispose of a view group, use the `GXDisposeViewGroup` function, described next.

GXDisposeViewGroup

You can use the `GXDisposeViewGroup` function to delete a view group object.

```
void GXDisposeViewGroup(gxViewGroup target);
```

target A reference to the view group.

DESCRIPTION

The `GXDisposeViewGroup` function deletes the view group, as well as all view devices and view ports that belong to that view group. If you create an offscreen view group with several view ports and view devices, you needn't dispose of those view ports and view devices when you are finished, as long as you dispose of the view group itself.

ERRORS, WARNINGS, AND NOTICES**Errors**

`invalid_viewGroup_reference`

SEE ALSO

For an example of the use of this function, see page 7-63.

For information about view groups, see “About View Group Objects” beginning on page 7-29.

Getting the View Ports and View Devices of a View Group

The functions described in this section allow you to find out what view ports and view devices belong to a view group.

GXGetViewGroupViewPorts

You can use the `GXGetViewGroupViewPorts` function to retrieve a list of the view ports that are associated with a view group object.

```
long GXGetViewGroupViewPorts(gxViewGroup source,
                             gxViewPort list[]);
```

`source` A reference to the view group whose view ports you wish to examine.

`list` An array of view port references. On return, the array contains a list of references to the view ports belonging to the source view group.

function result The number of view port references in the `list` array.

DESCRIPTION

The `GXGetViewGroupViewPorts` function fills out a list of all the view ports in the source view group and returns, as its function result, the number of view ports in the list.

If you pass `gxAllViewDevices` for the `source` parameter, this function returns all view ports in all view groups.

If you set the `list` parameter to `nil`, `GXGetViewGroupViewPorts` does not fill out the list of references; it only returns the number of view port references that would be in the list. Thus, you typically call this function twice: first to get the size of array to allocate for the `list` parameter, and second to retrieve the list itself.

ERRORS, WARNINGS, AND NOTICES

Errors

`invalid_viewGroup_reference`

SEE ALSO

To get a list of all the view devices in a view group, use the `GXGetViewGroupViewDevices` function, described next.

GXGetViewGroupViewDevices

You can use the `GXGetViewGroupViewDevices` function to retrieve a list of the view devices that are associated with a view group object.

```
long GXGetViewGroupViewDevices(gxViewGroup source,
                               gxViewDevice list[]);
```

`source` A reference to the view group whose view devices you wish to examine.
`list` An array of view device references. On return, the array contains a list of references to the view devices belonging to the source view group.

function result The number of view device references in the `list` array.

DESCRIPTION

The `GXGetViewGroupViewDevices` function fills out a list of all the view devices in the source view group and returns, as its function result, the number of view devices in the list.

If you pass `gxAllViewDevices` for the `source` parameter, this function returns all view devices in all view groups.

If you set the `list` parameter to `nil`, `GXGetViewGroupViewDevices` does not fill out the list of references; it only returns the number of view device references that would be in the list. Thus, you typically call this function twice: first to get the size of array to allocate for the `list` parameter, and second to retrieve the list itself.

ERRORS, WARNINGS, AND NOTICES**Errors**

`invalid_viewGroup_reference`

SEE ALSO

For an example of the use of this function, see Listing 7-10 on page 7-54.

To get a list of all the view ports in a view group, use the `GXGetViewGroupViewPorts` function, described in the previous section.

Measuring a Shape in Global Coordinates

The function described in this section allows you to determine the bounding rectangle of a shape in the global coordinates of its view group. Other QuickDraw GX functions are available to measure shapes in other contexts:

- n To determine a shape's bounding rectangle in local coordinates, use the `GXGetShapeLocalBounds` function, described on page 7-96.
- n To determine a shape's bounding rectangle on a view device, use the `GXGetShapeDeviceBounds` function, described on page 7-116.
- n To determine a shape's bounding rectangle in geometry-space coordinates, use the `GXGetShapeBounds` function, described in the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*.

GXGetShapeGlobalBounds

You can use the `GXGetShapeGlobalBounds` function to determine the bounding rectangle of a shape in global coordinates.

```
boolean GXGetShapeGlobalBounds(gxShape source,
                               gxViewPort port,
                               gxViewGroup group,
                               gxRectangle *bounds);
```

<code>source</code>	A reference to the shape whose bounding rectangle you wish to determine in global coordinates.
<code>port</code>	A reference to a view port to which the shape is drawn.
<code>group</code>	A reference to the view group that defines the global coordinates.
<code>bounds</code>	A pointer to a rectangle structure. On return, the structure contains the bounding rectangle for the shape, in the global coordinates of the specified view group.

function result `true` if the bounding rectangle appears in global space; `false` if it does not.

DESCRIPTION

The `GXGetShapeGlobalBounds` function returns a value that specifies whether the bounding rectangle of the shape drawn to the specified view port appears anywhere in the global space of the specified view group. The view port must belong to the view group, and it must be referenced in the view port list of the shape's transform object.

The `GXGetShapeGlobalBounds` function also returns in the `bounds` parameter the bounding rectangle of that part of the shape that can be drawn through the specified view port. The function returns the bounding rectangle after the shape's transform clip, mapping and style have been applied, and after all view port mappings and clips have been applied, from the view port specified in the `port` parameter to the root view port in the view port hierarchy (if any). The dimensions of the rectangle are in the global coordinates of the view group.

If you specify `nil` for the `port` parameter, `GXGetShapeGlobalBounds` includes all view ports specified in the source shape's transform's view port list. If you specify `nil` for the `group` parameter, `GXGetShapeGlobalBounds` includes all view groups of all specified view ports.

To determine a shape's bounding rectangle in geometry-space coordinates, use the `GXGetShapeBounds` function. To determine a shape's bounding rectangle in local coordinates, use the `GXGetShapeLocalBounds` function. To determine a shape's bounding rectangle in device coordinates, use the `GXGetShapeDeviceBounds` function.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>invalid_viewPort_reference</code>	
<code>invalid_viewGroup_reference</code>	
<code>parameter_is_nil</code>	(debugging version)

Notices (debugging version)

<code>transform_references_disposed_viewPort</code>

SEE ALSO

For an example of the use of this function, see Listing 7-15 on page 7-64.

The `GXGetShapeBounds` function is described in the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*. The `GXGetShapeLocalBounds` function is described on page 7-96. The `GXGetShapeDeviceBounds` function is described on page 7-116.

Summary of View-Related Objects

Constants and Data Types

The View Port Object

```
typedef struct gxPrivateViewPortRecord *gxViewPort;
```

The Halftone Structure

```
struct gxHalftone{
    Fixed          angle;
    Fixed          frequency;
    gxDotType      method;
    gxTintType     tinting;
    gxColor        dotColor;
    gxColor        backgroundColor;
    gxColorSpace   tintSpace;
};
```

Dot Types

```
enum gxDotTypes{
    gxRoundDot = 1,
    gxSpiralDot,
    gxSquareDot,
    gxLineDot,
    gxEllipticDot,
    gxTriangleDot,
    gxDispersedDot
};
```

```
typedef long gxDotType;
```

Tint Types

```
enum gxTintTypes{
    gxNoTint,
    gxLuminanceTint,
    gxAverageTint,
};
```

View-Related Objects

```

    gxMixtureTint,
    gxComponent1Tint,
    gxComponent2Tint,
    gxComponent3Tint,
    gxComponent4Tint
};

```

```
typedef long gxTintType;
```

View Port Attributes

```

enum gxPortAttributes {
    gxGrayPort          = 0x0001, /* convert to gray space */
    gxAlwaysGridPort    = 0x0002, /* use gxDeviceGridStyle */
    gxEnableMatchPort   = 0x0004 /* perform color matching */
};

```

```
typedef long gxPortAttribute;
```

The View Device Object

```
typedef struct gxPrivateViewDeviceRecord *gxViewDevice;
```

View Device Attributes

```

enum gxDeviceAttributes{
    gxDirectDevice      = 0x01, /* pixel image must be accessible */
    gxRemoteDevice      = 0x02, /* pixel image may be on card */
    gxInactiveDevice    = 0x04 /* device is inactive */
};

```

```
typedef long gxDeviceAttribute;
```

The View Group Object

```
typedef struct gxPrivateViewGroupRecord *gxViewGroup;
```

View Group Types

```

#define gxAllViewDevices      ((gxViewGroup) 0)
#define gxScreenViewDevices  ((gxViewGroup) 1)

```

View Port Functions

Creating and Manipulating View Port Objects

```

gxViewport GXNewViewport    (gxViewGroup group);
void GXDisposeViewport     (gxViewport target);
gxViewport GXCopyToViewport (gxViewport target, gxViewport source);
boolean GXEqualViewport    (gxViewport one, gxViewport two);

```

Manipulating View Port Object Properties

```

gxShape GXGetViewportClip  (gxViewport source);
void GXSetViewportClip    (gxViewport target, gxShape clip);
gxMapping *GXGetViewportMapping
                          (gxViewport source, gxMapping *map);
void GXSetViewportMapping (gxViewport target, const gxMapping *map);
gxMapping *GXGetViewportGlobalMapping
                          (gxViewport source, gxMapping *map);
long GXGetViewportDither  (gxViewport source);
void GXSetViewportDither (gxViewport target, long level);
boolean GXGetViewportHalftone
                          (gxViewport source, gxHalftone *data);
void GXSetViewportHalftone (gxViewport target, const gxHalftone *data);
Fixed GXGetHalftoneDeviceAngle
                          (gxViewDevice source, const gxHalftone *data);
gxViewport GXGetViewportParent
                          (gxViewport source);
void GXSetViewportParent (gxViewport target, gxViewport parent);
long GXGetViewportChildren (gxViewport source, gxViewport list[]);
void GXSetViewportChildren (gxViewport target, long count,
                          const gxViewport list[]);
gxViewGroup GXGetViewportViewGroup
                          (gxViewport source);
void GXSetViewportViewGroup (gxViewport target, gxViewGroup group);
gxPortAttribute GXGetViewportAttributes
                          (gxViewport source);
void GXSetViewportAttributes(gxViewport target,
                          gxPortAttribute attributes);
long GXGetViewportTags    (gxViewport source, long tagType, long index,
                          long count, gxTag items[]);

```

```
void GXSetViewPortTags      (gxViewPort target, long tagType, long index,
                             long oldCount, long newCount,
                             const gxTag items[]);
```

Retrieving the View Devices That Intersect a View Port

```
long GXGetViewPortViewDevices(gxViewPort source, gxViewDevice list[]);
```

Retrieving the View Ports That Intersect a Shape

```
long GXGetShapeGlobalViewPorts
                             (gxShape source, gxViewPort list[]);
```

Measuring a Shape in Local Coordinates

```
gxRectangle *GXGetShapeLocalBounds
                             (gxShape source, gxRectangle *bounds);
```

View Device Functions

Creating and Manipulating View Device Objects

```
gxViewDevice GXNewViewDevice(gxViewGroup group, gxShape bitmapShape);
void GXDisposeViewDevice    (gxViewDevice target);
gxViewDevice GXCopyToViewDevice
                             (gxViewDevice target, gxViewDevice source);
boolean GXEqualViewDevice   (gxViewDevice one, gxViewDevice two);
```

Manipulating View Device Object Properties

```
gxShape GXGetViewDeviceClip (gxViewDevice source);
void GXSetViewDeviceClip    (gxViewDevice target, gxShape clip);
gxMapping *GXGetViewDeviceMapping
                             (gxViewDevice source, gxMapping *map);
void GXSetViewDeviceMapping (gxViewDevice target, const gxMapping *map);
gxShape GXGetViewDeviceBitmap
                             (gxViewDevice source);
void GXSetViewDeviceBitmap  (gxViewDevice target, gxShape bitmapShape);
gxViewGroup GXGetViewDeviceViewGroup
                             (gxViewDevice source);
void GXSetViewDeviceViewGroup
                             (gxViewDevice target, gxViewGroup group);
gxDeviceAttribute GXGetViewDeviceAttributes
                             (gxViewDevice source);
```



```

void GXSetViewDeviceAttributes
    (gxViewDevice target, gxDeviceAttribute
     attributes);
long GXGetViewDeviceTags    (gxViewDevice source, long tagType, long index,
                             long count, gxTag items[]);
void GXSetViewDeviceTags    (gxViewDevice target, long tagType, long index,
                             long oldCount, long newCount,
                             const gxTag items[]);

```

Retrieving the View Devices That Intersect a Shape

```

long GXGetShapeGlobalViewDevices
    (gxShape source, gxViewPort port,
     gxViewDevice list[]);

```

Measuring a Shape in Device Coordinates

```

boolean GXGetShapeDeviceBounds
    (gxShape source, gxViewPort port,
     gxViewDevice device, gxRectangle *bounds);
long GXGetShapeDeviceArea    (gxShape source, gxViewPort port,
                              gxViewDevice device);

```

Measuring the Colors and Pattern Width of a Shape on a Device

```

gxColorSet GXGetShapeDeviceColors
    (gxShape source, gxViewPort port,
     gxViewDevice device, long *width);

```

Hit-Testing a Shape on a Device

```

gxShape GXHitTestDevice    (gxShape target, gxViewPort port,
                             gxViewDevice device, const gxPoint *test,
                             const gxPoint *tolerance);

```

View Group Functions

Creating and Disposing of View Group Objects

```

gxViewGroup GXNewViewGroup (void);
void GXDisposeViewGroup    (gxViewGroup target);

```

Getting the View Ports and View Devices of a View Group

```
long GXGetViewGroupViewPorts(gxViewGroup source, gxViewPort list[]);  
long GXGetViewGroupViewDevices  
    (gxViewGroup source, gxViewDevice list[]);
```

Measuring a Shape in Global Space

```
boolean GXGetShapeGlobalBounds  
    (gxShape source, gxViewPort port,  
     gxViewGroup group, gxRectangle *bounds);
```

Tag Objects

Contents

About Tag Objects	8-3
Tag Object Properties	8-4
Tag Types	8-5
Uses for Tag Objects	8-6
Using Tag Objects	8-7
Creating and Manipulating Tag Objects	8-7
Creating and Deleting a Tag Object	8-8
Copying, Comparing, and Cloning Tag Objects	8-9
Loading and Unloading Tag Objects	8-9
Manipulating Tag Object Properties	8-9
Getting and Setting a Tag Object's Tag Type and Contents	8-10
Manipulating a Tag Object's Owner Count	8-11
Directly Manipulating Tag Object Contents	8-11
Attaching Tags to a QuickDraw GX Object	8-12
Tag Objects Reference	8-12
Constants and Data Types	8-13
The Tag Object	8-13
Functions	8-13
Creating and Manipulating Tag Objects	8-13
GXNewTag	8-13
GXDisposeTag	8-14
GXCopyToTag	8-15
GXEqualTag	8-16
GXCloneTag	8-17
Manipulating Tag Object Properties	8-18
GXGetTag	8-18
GXSetTag	8-19
GXGetTagOwners	8-20

CHAPTER 8

Directly Manipulating the Data in a Tag Object	8-21
GXLockTag	8-21
GXUnlockTag	8-22
GXGetTagStructure	8-23
Summary of Tag Objects	8-25
C Summary	8-25
Functions	8-25

Tag Objects

This chapter describes tag objects and the functions you can use to manipulate them. Tag objects encapsulate application-defined information that can provide information about or modify the behavior of the QuickDraw GX objects associated with those tag objects.

Read this chapter if you need to create or modify tag objects. Other chapters in this book describe the functions you use to add tag objects to or delete tag objects from specific other kinds of QuickDraw GX objects, such as shapes or styles.

Before reading this chapter, you should be familiar with the information in the chapter “Introduction to QuickDraw GX” in this book.

This chapter introduces QuickDraw GX tag objects and describes their properties. It then shows how to use the QuickDraw GX tag-manipulation functions to

- n create and manipulate tag objects
- n manipulate tag object properties
- n directly manipulate tag contents

About Tag Objects

A tag object is a private data structure whose purpose is to allow any kind of application-defined information to be attached to a QuickDraw GX object. An object such as a shape or transform can be “tagged” with data or code that alters its behavior in specific situations or provides extra information about it. For example, you can attach identifying strings to objects with tags, or you can alter the way an object is displayed on a particular imaging device by attaching a tag to it that contains imaging commands specific to that device. For example, QuickDraw GX uses tag objects to hold PostScript commands used for printing to PostScript devices.

QuickDraw GX identifies an individual tag object through a tag **reference**. To obtain information about a tag object, you must send its reference as a parameter to a QuickDraw GX function (except that you can determine if two references identify the same tag object simply by comparing them for equality, and you can examine a reference to see if it is `nil`).

Tag objects are further identified by **tag type**, a designation that you can use to identify the tag object’s purpose and format.

A tag object is attached to its associated object by means of a **tag list**, a property that most QuickDraw GX objects have. A tag list is an array of references to the tag objects attached to an object. Objects can thus have more than one attached tag object. You cannot attach a tag object to a printing object, a font object, a graphics client object, or a tag object itself; there is no tag list property for those objects.

Tag Objects

Because tags are QuickDraw GX objects, they can be shared. A single tag object can be attached to more than one other object; the owner count of the tag object tells you how many references to it exist. Tags also have all the other advantages of QuickDraw GX objects: they are accessible from objects in accelerator memory, they can be unloaded to disk and reloaded automatically, and they can be flattened and included in a spool file.

Tag Object Properties

The interface to tag objects is entirely procedural. You manipulate the information in a tag object by modifying its properties using QuickDraw GX functions.

Tag objects have four accessible properties, as shown in Figure 8-1. Note that, because a tag is an object and not a data structure, the order of the properties as shown in Figure 8-1 is completely arbitrary.

Figure 8-1 The tag object and its properties



These are the four accessible properties of a tag object:

- n **Tag type.** A 4-byte value that specifies the type of this tag object. On the Macintosh computer, tag types are typically represented with four-character mnemonics, such as 'DAVE'.
- n **Size.** The size in bytes of the contents of the tag object.
- n **Contents.** The data that makes up this tag object. QuickDraw GX is unconcerned with the nature of the data; you can place whatever information you wish into the contents of a tag object.
- n **Owner count.** The number of existing references to this tag object.

QuickDraw GX provides functions to manipulate each of these tag object properties.

Tag Types

Tag objects have types in order to identify their purpose. For example, if you want to identify a circle that is approximated by a QuickDraw GX path shape, you might attach to it a tag of type 'CRCL'. Then, whenever your application scales a path shape, it can first check to see if there is a tag object of type 'CRCL' attached to that shape. If there is, your application can make sure that the scaling preserves the circularity of the result. If your application has its own circle-drawing function, it can call that function instead of calling `GXDrawShape` to draw the circle.

The creator of a tag object can give it any 4-byte type value, although it is customary to make it a value that can be represented with four 1-byte ASCII characters. Apple Computer, Inc., reserves all tag types that can be represented with lowercase characters only, such as 'dave'. There are no other restrictions, except that a tag type cannot be 0. If you intend your tag type to be exportable (usable by other applications), you can be certain that it will not conflict with other applications' tag types if you use your application's creator type, as registered with Macintosh Developer Technical Support, as your tag type.

Note

A four-character tag type is not portable. On systems other than the Macintosh, the tag type may print or display quite differently, and one with the same appearance may have a very different numeric value. For maximum portability, it is best to define tag types with hexadecimal values, such as

```
#define daveTag 0x44415645          /* 'DAVE' */
```

In this way, the tag type `daveTag` will be correct regardless of the architecture of the machine it is defined on. `u`

Some tag types have already been defined for specific purposes. QuickDraw GX uses tag objects for printing synonyms, which include data such as PostScript commands that replace the QuickDraw GX drawing commands for printing on PostScript printers. QuickDraw GX also uses tag objects to list fonts and individual glyphs used by flattened shapes. There are several currently defined tag types for printing synonyms, one for flattened fonts and glyphs, plus other tag types for various other purposes. Table 8-1 lists some of the currently defined tag types.

Tag Objects

Table 8-1 Defined tag types for tag objects

Tag type	Constant	Explanation
'flst'	gxFlatFontListItemTag	Tag object contains a list of fonts used by the associated object (a flattened shape).
'bfil'	gxBitmapFileAliasTagType	Tag object contains an alias record specifying the file that holds the pixel image for the associated bitmap shape object.
'post'	gxPostScriptTag	Tag object contains PostScript instructions replacing the information in the associated object.
'psct'	gxPostControlTag	Tag object contains a control flag plus font and encoding information for a PostScript printer.
'sdsh'	gxDashSynonymTag	Tag object contains dash information to be used by the PostScript setdash operator.
'lcap'	gxLineCapSynonymTag	Tag object contains cap information to be used by the PostScript setlinecap operator.
'half'	gxFormathalftoneTag	Tag object contains halftone information to be used by a PostScript printer.
'ptrn'	gxPatternSynonymTag	Tag object contains pattern information to be used by vector devices.
'cubx'	gxCubicSynonymTag	Tag object contains a cubic Bézier representation of a curve or path.

Uses for Tag Objects

Tags were originally devised as generalized, object-based equivalents to QuickDraw picture comments. Picture comments are used for sending PostScript commands during printing and for other purposes. A tag is like a structured comment: it has a specific type, it is attached to a specific item (an object), and it has a specific scope (that object).

Tag objects can, however, be used for more than picture comments. For example, tags can provide general information. For a large, complex document that can be represented as a single picture shape, it may be important to know what application originally created the shape, or what ranges of properties (colors, pixel depths, page sizes, and so on) may be found in it. The shape may contain one or more references to tag objects that hold that information.

Tag Objects

You can also use tags to attach identifying strings to objects, for debugging or other purposes. You could name shapes with strings like “oval” or “topographic contour 3242”; you could name ink objects with strings like “cobalt blue” or “blend mode.” You could also use tag objects to attach user comments or descriptions to shapes.

Tag objects may also provide alternate behavior for an object when it is used outside the QuickDraw GX environment. For example, QuickDraw GX uses tag objects to store PostScript commands for drawing shapes to PostScript printers.

If you want to be able to draw a shape object on a system that uses a different coordinate system from QuickDraw GX, you could calculate and store the alternate coordinates in a tag attached to the shape. If you are working in a completely different graphics system that is a superset of QuickDraw GX, you could store that system’s graphics information as tag objects attached to the QuickDraw GX objects you create.

IMPORTANT

In most cases, an application-created tag object cannot change the behavior of its associated object within the QuickDraw GX environment. No geometric operations, no drawing operations, and no testing operations (such as `GXEqualShape`) take the existence of tag objects into account. (One minor exception is `GXFlattenShape`; see its description in the chapter “Shape Objects” in this book. A second exception is that drawing a bitmap whose pixel image is disk-based requires QuickDraw GX to use information in a tag object.) Other than that, tag objects can alter behavior only where graphics operations are overridden (as in printing), or where your application itself changes an operation based on the contents of a tag object. s

Using Tag Objects

This section describes how to create and manipulate tag objects and their contents. It describes how you can

- n create and manipulate tag objects
- n manipulate tag object properties
- n directly manipulate tag contents

Creating and Manipulating Tag Objects

This section describes how you can create and interact with tag objects as whole entities. To manipulate tag object properties, use the functions described in the section “Manipulating Tag Object Properties” beginning on page 8-9.

Creating and Deleting a Tag Object

QuickDraw GX provides the `GXNewTag` function to allow you to create a new tag object. When you create the tag object, you provide its contents and you specify its tag type. Once you have created the tag object, you can attach it to any QuickDraw GX object (except another tag object) by making a call such as `GXSetShapeTags`, `GXSetInkTags`, or `GXSetColorProfileTags`.

Except when it overrides its own functions (as during printing), QuickDraw GX does not access or use the internal structure of the tag object you create; its contents and function are entirely up to you. Nor does QuickDraw GX make any restrictions on the tag type designation you provide, except that it cannot be zero. QuickDraw GX does not make any use of tag type except to use it for retrieving and replacing tag objects according to your instructions.

To delete your application's reference to a tag object, call the `GXDisposeTag` function. Calling `GXDisposeTag` may or may not actually release the memory allocated for the object, depending on the object's owner count. The function decreases the owner count of the tag object by 1; if that brings the owner count to zero, the object is completely deleted and its memory released. See "Manipulating a Tag Object's Owner Count" on page 8-11. Owner counts and what it means to dispose of an object are described in general in the chapter "Introduction to Objects" in this book.

Listing 8-1 is a library function that takes an arbitrary amount of data, makes it into a tag object of a given tag type, and attaches it to a specified shape. The function uses the `GXNewTag` function to create the tag object, and the `GXDisposeTag` function to dispose of its reference to the tag object after attaching it to the shape.

Listing 8-1 Adding data to a shape as a tag object

```
void AddShapeUser(gxShape source, const void *data,
                 long length, long type)
{
    gxTag tempItem;
    tempItem = GXNewTag(type, length, data);
    GXSetShapeTags(source, 0, 0, 0, 1, &tempItem);
    GXDisposeTag(tempItem);
}
```

The `GXNewTag` function is described on page 8-13. The `GXDisposeTag` function is described on page 8-14.

Copying, Comparing, and Cloning Tag Objects

You can use the `GXCopyToTag` function to copy the information from one tag object to another or to create a new copy of an existing tag object.

You can test if two references refer to the same tag object by simply testing the references for equality. You can also compare two different tag objects for equality with the `GXEqualTag` functions. For two tag objects to be equal, their tag types and contents must be identical, although their owner counts need not be.

Object copies created with the `GXCopyToTag` function are always equal, by the criteria of `GXEqualTag`, to the objects from which they were copied.

In certain circumstances, you may want to copy a reference to a tag object without actually copying the object. This is called cloning, and you can use the `GXCloneTag` function to clone a tag object. Functionally, `GXCloneTag` does nothing more than increase the owner count of the tag object. For more information about cloning objects, see the chapter “Introduction to Objects” in this book. For information on manipulating owner counts, see the section “Manipulating a Tag Object’s Owner Count” on page 8-11.

The `GXCopyToTag` function is described on page 8-15. The `GXEqualTag` function is described on page 8-16. The `GXCloneTag` function is described on page 8-17.

Loading and Unloading Tag Objects

Although you rarely need to, you can influence memory-allocation decisions involving objects that you have created. If your application needs to have a tag object in memory, you can force QuickDraw GX to load the tag object into memory. When your application no longer needs the tag object in a loaded state, you can instruct QuickDraw GX to unload it.

You call the `GXLoadTag` function to make sure that a tag object is in memory; if necessary, QuickDraw GX brings the object into memory from an unloaded state. You can call the `GXUnloadTag` function to instruct QuickDraw GX that it is free to unload the tag object at any time. These functions are described in the memory management chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

Manipulating Tag Object Properties

This section describes how to manipulate the object properties of tag objects: tag type, contents, and owner count. To create and interact with tag objects as whole entities, use the functions described in the previous section, “Creating and Manipulating Tag Objects” beginning on page 8-7.

Getting and Setting a Tag Object's Tag Type and Contents

Fundamentally, tag objects are nothing but holders for whatever information or data that you want to attach to a QuickDraw GX object. If you want to access that information to inspect it or modify it, you can set up a buffer and call the `GXGetTag` function for a given tag object. `GXGetTag` places a copy of the tag object's information in your buffer, and it also returns the tag type of the tag object.

You can then modify the information in the buffer in any way you need to, and you can also change the tag type of the object, if desired. (Tag types that can be represented as four lowercase characters, such as 'abcd', are reserved by Apple Computer, Inc.) To return the modified contents or tag type to the tag object they came from, you then call the `GXSetTag` function.

Listing 8-2 is a library function that retrieves the data of the first tag object of a specified tag type or index attached to a given shape. If you specify an index and a buffer length to put the data in, the function returns the contents and tag type of the found tag. The function result is the number of tags found that fit the criteria you specify. The function uses the `GXGetTag` function to retrieve the tag object's contents.

Listing 8-2 Retrieving the contents of a tag object

```
long GetShapeUser(gxShape source, void *data, long *length, long
requestedType, long *foundType, long index)
{
    if( index )          /* if index nonzero, get specific tag */
    {
        gxTag tempItem;

        if( GXGetShapeTags(source, requestedType,
                           index, 1, &tempItem) )
        {
            long tempLength = GXGetTag(tempItem, foundType, data);
            if( length )
                *length = tempLength;
            return 1;          /* no. of tags found */
        } else
            return 0;
    }
    else                  /* otherwise just get tags of req. type */
        return GXGetShapeTags(source, requestedType,
                               1, gxSelectToEnd, nil);
}
```

The `GXGetTag` function is described on page 8-18.

Tag Objects

Using the `GXGetTag` and `GXSetTag` functions involves working with a copy of the tag object's contents in a buffer in application memory. You can also gain direct access to a tag object's contents (but not tag type) in QuickDraw GX memory, if desired. See the section "Directly Manipulating Tag Object Contents" beginning on page 8-11.

Manipulating a Tag Object's Owner Count

The owner count of an object indicates the number of current references to that object. In general, QuickDraw GX manages owner counts for you. For example, when you create a new tag object, QuickDraw GX sets the owner count of the new tag object to 1, which corresponds to the variable your application uses to reference the tag object. As another example, when you assign an existing tag object to a shape or transform (or any other object), QuickDraw GX increments the tag object's owner count, corresponding to the new reference to the tag object contained in the style or transform.

If you want to directly manage the owner count of a tag object yourself, or if you want to know whether a tag object is shared, you can

- n use the function `GXGetTagOwners` to determine the current owner count
- n use the function `GXCloneTag` to increment the owner count, whenever you create a new reference to the object
- n use the function `GXDisposeTag` to decrement the owner count, deleting the tag object and freeing the memory used by it if the owner count goes to 0

The `GXGetTagOwners` function is described on page 8-20.

Note

In the chapter "Style Objects" in this book, the section on manipulating a style object's owner count discusses two common owner-count problems and how to avoid them. The problems are discussed in terms of style objects, but they apply equally well to tag objects. Refer to that discussion if you find that tag objects you create have owner counts that are higher or lower than you expect. u

Directly Manipulating Tag Object Contents

Unlike with most properties of most objects, QuickDraw GX allows you to directly manipulate a tag object's contents in QuickDraw GX memory. This capability is provided as a convenience, so that you do not have to make a copy of the data; you can achieve the same results by working with a copy of the information in application memory and then replacing it in the object, using the `GXGetTag` and `GXSetTag` functions. See "Getting and Setting a Tag Object's Tag Type and Contents" on page 8-10 for information on working with `GXGetTag` and `GXSetTag`.

As with `GXGetTag` and `GXSetTag`, the direct-manipulation functions do not provide you with information about the format or organization of the contents of a tag object; they simply give you a pointer to the contents. How you manipulate that information depends on the type of tag you are accessing.

Tag Objects

Working with data in QuickDraw GX memory requires that you lock the data before accessing it so that it cannot be relocated or unloaded from memory until you are finished. You first call the `GXLockTag` function to make sure the tag object doesn't move until you are finished with it. You then call the `GXGetTagStructure` function, which returns a pointer to and the size of the shape's contents. You can then modify the data as needed. Once finished, you call `GXUnlockTag` to free the tag object for relocation as needed by QuickDraw GX.

IMPORTANT

Memory-handling complications can occur with locked objects. Locking an object fragments the QuickDraw GX heap, which can result in lower performance. Furthermore, if a fragmented-memory condition occurs during a call, QuickDraw GX may unlock all objects and restart the call. Therefore, be careful about performing memory-intensive operations while there are locked objects in QuickDraw GX memory; they may become unlocked without warning. [s](#)

The `GXLockTag` function is described on page 8-21. The `GXGetTagStructure` function is described on page 8-23. The `GXUnlockTag` function is described on page 8-22.

Attaching Tags to a QuickDraw GX Object

Most QuickDraw GX objects (other than tag objects themselves) can have one or more attached tag objects. Each object has a property called a **tag list**, which is an array of references to tag objects. You can retrieve and assign tag references with QuickDraw GX functions. For example, to retrieve one or more of the tags attached to a shape object, you call the `GXGetShapeTags` function. To add one or more tags to a shape object's tag list, you call the `GXSetShapeTags` function.

Besides tag objects, other objects that you cannot directly attach tags to include printing objects, font objects, and graphics client objects.

The `GXGetShapeTags` and `GXSetShapeTags` functions are described in the chapter "Shape Objects" in this book; the equivalent calls for other kinds of objects are described in the chapters that document those objects.

Tag Objects Reference

This section provides reference information to the data structures and functions that allow you to create and manipulate tag objects and alter their properties.

Constants and Data Types

This section describes the data type that you use to gain access to tag objects.

The Tag Object

QuickDraw GX provides you with access to an individual tag object through a `gxTag` reference:

```
typedef struct gxPrivateTagRecord *gxTag;
```

In this type definition, `gxTag` is a type-checked reference, not an actual pointer to any defined structure. The contents of the tag object are private.

Functions

This section describes the functions with which you can

- n create and manipulate tag objects
- n manipulate tag object properties, including the contents of the tag object
- n directly manipulate the contents of a tag object, in-place in QuickDraw GX memory

Creating and Manipulating Tag Objects

The functions in this section allow you to create and manipulate tags as QuickDraw GX objects.

GXNewTag

You can use the `GXNewTag` function to create a new tag object.

```
gxTag GXNewTag(long tagType, long length, const void *data);
```

`tagType` A 4-byte identifier specifying the type of tag object to be created.

`length` The length in bytes of the data to place in the tag object.

`data` A pointer to the data to place in the tag object.

function result A reference to the newly created tag object.

DESCRIPTION

The `GXNewTag` function creates a tag object, with an owner count of 1, containing whatever data you supply. The tag type is an application-defined, 4-byte tag (commonly expressed on the Macintosh with four characters, such as 'Cary') that you supply to specify the type of data contained in the tag object. You cannot supply a value of 0.

You can specify a value of zero for the `length` parameter, in which case `GXNewTag` creates the tag object but places no data into it. (In that case, the `data` pointer must be `nil`, or else `GXNewTag` posts an `inconsistent_parameters` error.)

SPECIAL CONSIDERATIONS

You cannot specify a tag type of zero. Apple Computer, Inc., reserves all tag types that can be expressed as four lowercase characters (such as 'atag').

Tag types expressed as four characters may not be portable to systems other than the Macintosh; tag types defined numerically are portable.

If no error occurs, the `GXNewTag` function creates a tag object; you are responsible for disposing of that object when you no longer need it.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`type_is_nil`
`inconsistent_parameters`

SEE ALSO

Currently defined tag types for tag objects are listed in Table 8-1 on page 8-6.

GXDisposeTag

You can use the `GXDisposeTag` function to release a reference to a tag object.

```
void GXDisposeTag(gxTag target);
```

`target` A reference to the tag object to dispose of.

DESCRIPTION

The `GXDisposeTag` function decrements the owner count of the tag object specified by the `target` parameter. `GXDisposeTag` deletes the tag object and releases any memory used by it if the owner count goes to zero.

SPECIAL CONSIDERATIONS

If you attempt to alter a tag object associated with a screen view device, this function posts a `tag_access_restricted` error.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>tag_is_nil</code>	
<code>cannot_dispose_locked_tag</code>	(debugging version)
<code>tag_access_restricted</code>	(debugging version)

SEE ALSO

Owner counts for tag objects are discussed in the section “Copying, Comparing, and Cloning Tag Objects” beginning on page 8-9, and in the section “Manipulating a Tag Object’s Owner Count” beginning on page 8-11.

To examine the owner count of a tag object, use the `GXGetTagOwners` function, described on page 8-18. To increment the owner count of a tag, use the `GXCloneTag` function, described on page 8-17.

GXCopyToTag

You can use the `GXCopyToTag` function to copy the contents of one existing tag object into another, or to create a new tag object and copy the contents of an existing tag object into it.

```
gxTag GXCopyToTag(gxTag target, gxTag source);
```

<code>target</code>	A reference to the tag object to copy into. If you specify <code>nil</code> for this parameter, the <code>GXCopyToTag</code> function creates a new tag object.
<code>source</code>	A reference to the tag object whose contents you want to copy.

function result A reference to the copy (that is, the target tag object).

DESCRIPTION

The `GXCopyToTag` function copies the contents of an existing tag object to another, or it creates a new tag object and copies the contents of an existing tag object to it. The function copies the tag type and contents (but not the owner count) of the tag object specified by the `source` parameter into the tag object specified by the `target` parameter.

If you specify `nil` for the `target` parameter, the `GXCopyToTag` function creates a new tag object and copies the source tag object's tag type, contents, and owner count into it.

You can use the `GXCopyToTag` function to create a copy of a tag object and then modify it without changing the original.

SPECIAL CONSIDERATIONS

If you specify `nil` for the `target` parameter and no error occurs, the `GXCopyToTag` function creates a tag object; you are responsible for disposing of that object when you no longer need it.

If you attempt to alter a tag object associated with a screen view device, this function posts a `tag_access_restricted` error.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`tag_is_nil`
`tag_access_restricted` (debugging version)

SEE ALSO

To create a new tag object that is not a copy of an existing tag object, use the `GXNewTag` function, described on page 8-13.

To compare two tag objects, use the `GXEqualTag` function, described in the next section.

GXEqualTag

You can use the `GXEqualTag` function to determine whether two tag objects are equal.

```
boolean GXEqualTag(gxTag one, gxTag two);
```

`one` A reference to one of the tag objects to test for equality.

`two` A reference to the other tag object to test for equality.

function result `true` if the two tag objects are equal; `false` otherwise.

DESCRIPTION

The `GXEqualTag` function returns `true` if the two specified tag objects are equal. For two tag objects to be equal, they must have identical tag types and contents, although their owner counts need not be identical.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`tag_is_nil`

SEE ALSO

To make a copy of a tag object that is equal by the criteria of this function, use the `GXCopyToTag` function, described in the previous section.

GXCloneTag

You can use the `GXCloneTag` function to clone a tag object—that is, to add a reference to it and increment its owner count.

```
gxTag GXCloneTag(gxTag source);
```

`source` A reference to the tag object to clone.

function result A reference to the cloned tag object.

DESCRIPTION

The `GXCloneTag` function increments the owner count of the tag object referenced in the `source` parameter. You typically use this function when you want to create another reference to an existing tag object instead of creating a distinct copy of the tag.

This function returns as its function result a reference to the tag object—the same reference you pass in as the `source` parameter. It also increments the tag object's owner count.

ERRORS, WARNINGS, AND NOTICES**Errors**

`tag_is_nil`
`tag_access_restricted` (debugging version)

SEE ALSO

Owner counts for tag objects are discussed in the section “Copying, Comparing, and Cloning Tag Objects” beginning on page 8-9, and in the section “Manipulating a Tag Object’s Owner Count” beginning on page 8-11.

To examine the owner count of a tag object, use the `GXGetTagOwners` function, described on page 8-20. To decrement the owner count of a tag, use the `GXDisposeTag` function, described on page 8-14.

Manipulating Tag Object Properties

The functions in this section allow you to manipulate the object properties of tag objects: their tag types, contents, and owner counts.

GXGetTag

You can use the `GXGetTag` function to retrieve the tag type and contents of a tag object.

```
long GXGetTag(gxTag source, long *tagType, void *data);
```

<code>source</code>	A reference to the tag object whose contents you want to retrieve.
<code>tagType</code>	A pointer to a value specifying a tag object type. On return, specifies the type of tag object referenced in the <code>source</code> parameter.
<code>data</code>	A pointer to a buffer. On return, the buffer holds the contents of the source tag object.

function result The size in bytes of the contents of the source tag object.

DESCRIPTION

The `GXGetTag` function returns the tag type and contents of the source tag object in the `tagType` and `data` parameters, respectively. Its function result is the size of the information returned in the `data` array.

Before calling `GXGetTag`, you must allocate an array of sufficient size to hold the contents of the tag object. If instead you pass `nil` for the `data` parameter, `GXGetTag` does not return the tag contents, but nonetheless returns (as its function result) the size of the contents. Thus you can make an initial call to `GXGetTag` to determine the size of buffer to allocate, and then call `GXGetTag` once more to get the contents.

The `GXGetTag` function is different from the `GXGetTagStructure` function in that it returns a copy of the tag object’s contents in a buffer that you have allocated in application memory. The `GXGetTagStructure` gives you direct access to the contents of a tag object in QuickDraw GX memory.

Tag Objects

Although `GXGetTag` returns the contents of a tag object, it returns no information other than size about the format or organization of the tag's contents. You must know the internal structure of a tag object's contents in order to manipulate it.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`tag_is_nil`

SEE ALSO

To replace the tag type and contents of a tag object, use the `GXSetTag` function, described in the next section.

The `GXGetTagStructure` function is described on page 8-23.

GXSetTag

You can use the `GXSetTag` function to replace the tag type or contents of a tag object.

```
void GXSetTag(gxTag target, long tagType, long length,
             const void *data);
```

<code>target</code>	A reference to the tag object whose contents you want to replace.
<code>tagType</code>	The new tag type to assign to the tag object referenced in the <code>target</code> parameter. If you pass 0 for this parameter, the tag type remains unchanged.
<code>length</code>	The length in bytes of the new data to place in the tag object. If you pass 0 for this parameter, the contents of the tag object remain unchanged and the <code>data</code> parameter is ignored.
<code>data</code>	A pointer to the new data to place in the tag object. If you pass <code>nil</code> for this parameter, the contents of the tag object (up to the length specified by <code>length</code>) remain unchanged.

DESCRIPTION

The `GXSetTag` function assigns the specified tag type and contents to the target tag object. You can set three of its parameters for different purposes:

- n To change only the tag type and not the contents of a tag object, pass 0 in the `length` parameter, `nil` in the `data` parameter, and a nonzero value for `tagType`.
- n To change only the contents and not the tag type, pass 0 in the `tagType` parameter, and valid values for `length` and `data`.

Tag Objects

- n To resize the tag object without changing its contents or type, pass the new size in the `length` parameter, `nil` in the `data` parameter, and `0` in the `tagType` parameter. If the new size of the contents is smaller than the previous size, the data is truncated to fit the new size. If the new size is greater than the previous size, the tag object is resized accordingly, but a `new_tag_contains_invalid_data` warning is posted.

Note that calling `GXSetTag` is different from using the `GXGetTagStructure` function to manipulate the contents of a tag object. Unlike `GXGetTagStructure`, `GXSetTag` allows you to change the size of the tag object.

ERRORS, WARNINGS, AND NOTICES

Errors

<code>out_of_memory</code>	
<code>tag_is_nil</code>	
<code>inconsistent_parameters</code>	(debugging version)
<code>tag_access_restricted</code>	(debugging version)

Warnings

<code>new_tag_contains_invalid_data</code>
--

SEE ALSO

To retrieve the tag type and contents of a tag object, use the `GXGetTag` function, described in the previous section.

To directly manipulate the contents of a tag object in QuickDraw GX memory, rather than replacing its entire contents and tag type, use the `GXGetTagStructure` function, described on page 8-23.

GXGetTagOwners

You can use the `GXGetTagOwners` function to determine the number of references to a particular tag object.

```
long GXGetTagOwners(gxTag source);
```

`source` A reference to the tag object to find the owner count of.

function result The owner count of the tag object referenced in the `source` parameter.

DESCRIPTION

The `GXGetTagOwners` function returns the owner count of the referenced tag object. The owner count is the current number of references to the tag object.

ERRORS, WARNINGS, AND NOTICES**Errors**

tag_is_nil

SEE ALSO

Owner counts for tag objects are discussed in the section “Copying, Comparing, and Cloning Tag Objects” beginning on page 8-9, and in the section “Manipulating a Tag Object’s Owner Count” beginning on page 8-11.

To increment the owner count of a tag, use the `GXCloneTag` function, described on page 8-17. To decrement the owner count of a tag, use the `GXDisposeTag` function, described on page 8-14.

Directly Manipulating the Data in a Tag Object

This section describes the functions you use to directly manipulate the contents of a tag object in QuickDraw GX memory.

GXLockTag

You can use the `GXLockTag` function to load a tag object into QuickDraw GX memory and lock its contents into a fixed memory location.

```
void GXLockTag(gxTag target);
```

`target` A reference to the tag object to be loaded and locked.

DESCRIPTION

The `GXLockTag` function prevents a tag object from being relocated. To directly edit a tag’s contents, you must first call `GXLockTag`. You can then call `GXGetTagStructure` and edit the tag’s contents. After editing, you must call `GXUnlockTag` to release the tag for relocation.

SPECIAL CONSIDERATIONS

To avoid fragmenting the QuickDraw GX heap, you should call the `GXUnlockTag` function as soon as possible after calling `GXLockTag`.

You can nest calls to these direct-access routines, but be sure to call `GXUnlockTag` as many times as you call `GXLockTag`. Version 1.0 of QuickDraw GX prohibits more than 255 nested calls to `GXLockTag`.

ERRORS, WARNINGS, AND NOTICES**Errors**

out_of_memory
tag_is_nil

SEE ALSO

The `GXUnlockTag` function is described in the next section. The `GXGetTagStructure` function is described on page 8-23.

GXUnlockTag

You can use the `GXUnlockTag` function to allow QuickDraw GX to relocate, flatten, or unload a tag object.

```
void GXUnlockTag(gxTag target);
```

target A reference to the tag object to unlock.

DESCRIPTION

The `GXUnlockTag` function frees a previously locked tag object for relocation. To directly edit a tag's contents, you must first call `GXLockTag`. You can then call `GXGetTagStructure` and edit the tag's contents. After editing, you must call `GXUnlockTag`.

You cannot dispose of a tag that is locked. You must first call `GXUnlockTag` on a locked tag object before calling `GXDisposeTag`.

SPECIAL CONSIDERATIONS

To avoid fragmenting the QuickDraw GX heap, you should call the `GXUnlockTag` function as soon as possible after calling `GXLockTag`.

You can nest calls to these direct-access routines, but be sure to call `GXUnlockTag` as many times as you call `GXLockTag`.

RESULT CODE**Errors**

tag_is_nil

Notices (debugging version)

tag_not_locked

SEE ALSO

The `GXLockTag` function is described in the previous section. The `GXGetTagStructure` function is described in the next section.

The `GXDisposeTag` function is described on page 8-14.

GXGetTagStructure

You can use the `GXGetTagStructure` function to get a pointer to the contents of a tag object.

```
void *GXGetTagStructure(gxTag source, long *length);
```

`source` A reference to the tag object whose contents you need access to.

`length` A pointer to a value. On return, contains the size in bytes of the contents of the tag object referenced in the `source` parameter.

function result A pointer to the contents of the source tag object.

DESCRIPTION

The `GXGetTagStructure` function returns a pointer to the contents of the tag object referenced in the `source` parameter. To directly edit a tag's contents, you must first call `GXLockTag`. You can then call `GXGetTagStructure` and edit the tag's contents. After editing, you must call `GXUnlockTag`.

The `GXGetTagStructure` function is different from the `GXGetTag` function in that it gives you direct access to the contents of a tag object in QuickDraw GX memory. The `GXGetTag` function returns a copy of the tag object's contents in a buffer that you have allocated in application memory.

To edit the contents of a tag object, you need to know its format and organization. `GXGetTagStructure` returns a pointer and a size only; it does not provide you with any information about the internal structure of the tag's contents.

This function does not provide access to tag type information.

SPECIAL CONSIDERATIONS

Note that using the `GXGetTagStructure` is different from calling `GXSetTag`, in that it does not allow you to change the size of the tag object.

This function is available for your convenience, in that you do not have to make a copy of the tag object's data, but is rarely needed. In most cases you can use the `GXGetTag` and `GXSetTag` functions to manipulate tag contents.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`

`tag_is_nil`

Notices (debugging version)

`lockTag_called_as_side_effect`

SEE ALSO

The `GXLockTag` function is described on page 8-21. The `GXUnlockTag` function is described in the previous section.

The `GXGetTag` function is described on page 8-18. The `GXSetTag` function is described on page 8-19.

Summary of Tag Objects

C Summary

Functions

Creating and Manipulating Tag Objects

```
gxTag GXNewTag          (long tagType, long length, const void *data);
void GXDisposeTag      (gxTag target);
gxTag GXCopyToTag      (gxTag target, gxTag source);
boolean GXEqualTag     (gxTag one, gxTag two);
gxTag GXCloneTag       (gxTag source);
```

Manipulating Tag Object Properties

```
long GXGetTag          (gxTag source, long *tagType, void *data);
void GXSetTag          (gxTag target, long tagType, long length,
                      const void *data);
long GXGetTagOwners    (gxTag source);
```

Directly Manipulating the Data in a Tag Object

```
void GXLockTag        (gxTag target);
void GXUnlockTag      (gxTag target);
void *GXGetTagStructure (gxTag source, long *length);
```


Glossary

add mode A transfer mode type in which the source color component is added to the destination component, but the result is not allowed to exceed the maximum value (0xFFFF).

alignment A style object property. It is the alignment value to use when drawing the text of a shape. Text may be left-aligned, right-aligned, anywhere in the continuum between the two alignments (such as centered), or fully justified. (Layout shapes support continuous justification as well as continuous alignment.)

alpha channel A color component in some color spaces whose value represents the opacity of the color defined in the other components.

alpha-channel transfer modes Transfer mode types in which the result color is achieved by considering the alpha channel values as well as the color-component values of the source and destination.

AND mode A transfer mode type in which the bits of the source color component and destination color component are combined using an AND operation.

angle The angle from horizontal made by the pattern of dots in a halftone.

anti-aliasing The smoothing of jagged edges on a displayed shape by modifying the transparencies of individual pixels along the shape's edge.

application heap or **application memory** The part of computer memory directly accessible by an application, and in which its code and data structures reside. Compare **QuickDraw GX memory**, **graphics client heap**.

arithmetic transfer modes Transfer mode types in which the result color is achieved by using arithmetic operations on the source and destination color-component values.

atop mode A transfer mode type in which the source color is placed over the destination, but the resulting destination retains the original destination's transparency.

attributes A property of many QuickDraw GX objects. It is a set of flags that control various aspects of that object's behavior.

background color The color of the area between the dots of a halftone.

base family A group of closely related color spaces, across which color conversion can take place without the use of color profiles. RGB and HSV color spaces, for example, are all in the RGB base family.

bitmap shape A shape type that represents a pixel image. The geometry of a bitmap shape includes a **bitmap structure**.

bitmap structure A data structure that describes a pixel image.

black generation In CMYK color calculation, the substitution of black ink for areas with high intensities of cyan, magenta, and yellow. See also **undercolor removal**.

blend mode A transfer mode type in which the result is the average of the source and destination color components, weighted by a ratio specified by the operand component.

Boolean transfer modes Transfer mode types in which the result color is achieved by using Boolean operations on the bits of the source and destination color-component values.

cap A style object property. It is the shape (such as an arrowhead, or any other geometric shape) to draw at the start and end of each contour in the shape.

child view port For a given view port, a view port immediately below it in the view port hierarchy.

child view port list A view port object property. A view port's child view port list is an array of references to the child view ports of that view port.

chromaticity An intensity-independent color designation, represented by a pair of values (chromaticity coordinates) for the x and y components in Yxy space.

CIE Commission Internationale d'Eclairage, an organization that carried out experimental work that resulted in the definition of the XYZ and Yxy color spaces.

clip A property of a transform object, view port object, or view device object. It is a primitive shape, bitmap shape, or glyph shape that controls the visibility of a shape object.

clone To create another current reference to an existing object. The effect of cloning an object is to increase its owner count by 1.

closed-frame fill A shape fill that connects the points of the geometry from the start point through the end point and on to the start point again. Same as **hollow fill**.

CMYK color space A color space whose four components measure the cyan, magenta, yellow, and black elements of a color. Used mostly for printing.

Collection Manager A part of system software, related to QuickDraw GX, that manages **collection objects**.

collection object A kind of object, managed by the Collection Manager, that is used to hold any kind of information. Several QuickDraw GX printing objects have properties that are references to collection objects.

color A QuickDraw GX data structure—also a property of an ink object—that specifies a color in terms of a particular color space and the values for each of the color's components within that color space. A color structure can also contain a reference to a color profile object.

color-average tint A halftone tint type in which the tint color is specified by the average of all the components of the input color.

color component An individual dimension, or component, of a color space. For example, RGB space has three components: red, green, and blue.

color-component value A value representing the intensity of a single color component.

color conversion The conversion of a color value from one color space to another. If the color spaces are not in the same base family, accurate color conversion requires **color matching**.

colorimetric matching A color-matching method in which colors common to the gamuts of both devices are maintained across the match. Compare **perceptual matching**, **saturation matching**.

color-mixture tint A halftone tint type in which the tint color is specified by the mixture of dot color and background color closest to the input color.

color matching A method of accurately converting colors in one color space to another color space, or from display on one device to display on another device. Color matching requires the use of **color profiles** and a **color-matching method**.

color-matching method A specific algorithm for matching colors. Different algorithms have different purposes. See, for example, **colorimetric matching**, **perceptual matching**, **saturation matching**.

color packing The storing of colors in formats that are smaller than the unpacked formats. Whereas unpacked colors may require 48 or 64 bits to describe a color value, packed formats may require only 16 or 32 bits.

color profile A QuickDraw GX object associated with a transfer mode, color, or bitmap data structure and used for color matching. A color profile usually describes the color response curve of a display device in terms of an objective standard.

color set A QuickDraw GX object associated with a transfer mode or bitmap data structure. A color set defines the individual colors available for drawing a shape.

color space A specification of a particular method for color representation, such as RGB or HSV. QuickDraw GX recognizes over 30 different color space definitions.

ColorSync Utilities A part of Macintosh system software that manages color matching, color profiles, and the drawing of matched colors. QuickDraw GX color profile objects contain ColorSync color profiles, and QuickDraw GX uses the Color Sync Utilities to perform its color matching.

color value A structure that holds the full specification of a single color in a particular color space. For example, an RGB color value consists of three **color-component values**: one each for red, green, and blue. A color value is itself a component of a **color** structure.

color-value array A property of a color set object; it is the array of color values that constitute the colors of the color set.

component See **color component**.

component mode A transfer mode type, as applied to a single color component. It is the specification of the kind of transfer mode—such as copy mode or XOR mode—to apply to that color component when drawing a shape or pixel.

component tint A halftone tint type in which the tint color is specified by the value of one component of the input color.

concatenate To add (through matrix multiplication) the effects of one mapping matrix to another, as when the mappings of view ports in a view port hierarchy are concatenated to convert from local space to global space.

constructive geometry Mathematical operations, such as intersection and union, that construct new shape geometries out of input shape geometries.

coordinate plane See **coordinate space**.

coordinate space or **coordinate system** A planar region defined by all possible values for a pair of fixed-point coordinates. The coordinate spaces supported by QuickDraw GX include **geometry space**, **local space**, **global space**, and **device space**.

copy mode A transfer mode type in which the source color component is copied to the destination, and the destination component is ignored.

curve error A style object property. It is the allowable error for operations such as converting a path shape to a polygon shape.

curve shape A shape type that represents a quadratic Bézier curve.

dash A style object property. It is the appearance of dashed lines or contours in a shape. The dashing capability is very general in QuickDraw GX; you can specify any geometric shape, or even a sequence of glyphs, for a dash.

direct mode A fast printing mode that uses information built into the printer.

default object (1) For most kinds of objects, an object with the properties of a newly created object. When it creates an object, QuickDraw GX assigns it the default properties for that kind of object. (2) For color sets, the color set to assign as the default to a bitmap shape of a given pixel depth. (3) For color profiles, the profile to use for color matching when no profile is specified.

desktop printer A printer accessible through an icon on the user's desktop. The user prints to a desktop printer by dragging the icon of a document to the printer icon.

despool To open a print file and send its data to a device for printing. Compare **spool**.

destination color The preexisting color of the destination onto which a shape or pixel is to be drawn. Compare **source color**, **result color**.

destination color limits In a transfer mode, limits on the permissible values for destination color to use in transfer-mode calculations. Compare **source color limits**, **result color limits**.

destination profile The color profile attached to the device on which a color is to be drawn. Compare **source profile**.

device coordinates Paired values that specify a size or location in device space.

device matrix A 5×4 matrix, part of the transfer mode structure, that allows you to manipulate the components of the destination color.

device space The coordinate system that defines the position and scale (pixel size) of a specific view device. Compare **geometry space**, **local space**, **global space**.

direct memory Memory directly addressable by an application or by QuickDraw GX. Compare **remote memory**.

dispose To delete a reference to an object. When an application no longer needs an object, it disposes of the object. That action deletes the object from memory if there are no other current references to the object; otherwise, disposing of an object merely decreases its owner count by 1.

dither or **dither level** A property of a view port object. It specifies the number of colors that can be dithered together when drawing a shape to that view port.

dithering A color-display technique in which different colors are placed in adjacent pixels to achieve the affect of a single color intermediate between the displayed colors.

dot color The color of the dots of a halftone.

dot type The shape of dot employed in a halftone pattern, such as round, line, or triangle.

empty shape A shape type that has no geometry, no contents, and no bounds.

encoding A style object property. It is the type of character encoding used to represent the text of a shape, as well as its script and language.

error A diagnostic message posted by QuickDraw GX when a function cannot complete successfully.

error diffusion A process of dithering for bitmaps in which the error (the difference between the computed color of a given pixel and the nearest color available on the view device) is passed to adjacent pixels.

even-odd fill A shape fill that follows the even-odd rule. Same as **solid fill**.

exclude mode A transfer mode type in which the destination color remains visible only where the source is transparent, and the source color is copied anywhere the destination is transparent.

fade mode A transfer mode type in which the source is blended with the destination, using the relative alpha values as the ratio for the blend.

fill See **shape fill**.

flatten To convert the private, object-based description of an object or set of objects into a public-format data stream suitable for file or clipboard storage. Compare **unflatten**; see also **stream format**.

font A style object property. It is the reference to the font to use in drawing the text of a shape.

font variations A style object property. It is the list of font variations—stylistic variations built into the font—available for drawing the text of a shape.

format object A printing object that specifies page-formatting characteristics.

framed fill See **open-frame fill**.

frequency The size of cells in a halftone pattern, in cells per inch.

full shape A shape type that represents a shape that encompasses all of coordinate space.

fully justified See **justification**.

gamut The limits of the colors that a device can produce. Different devices have different gamuts, so color matching is necessary when converting colors from one device to another.

geometric operations Mathematical operations on the geometries of shape objects. See also **constructive geometry**.

geometry A property of a QuickDraw GX shape object. It is the specification of the actual size, position, and form of the shape. For example, for a rectangle shape, the geometry specifies the locations (in local coordinates) of the rectangle's upper-left and lower-right corners.

geometry coordinates Paired values that specify a size or location in geometry space.

geometry space The coordinate system represented by the geometry of a shape object. Compare **local space**, **global space**, **device space**.

global coordinates Paired values that specify a size or location in global space.

global space The coordinate system, used by a view group, resulting from the application of the view port mapping shape dimensions measured in local space. A view port's location, for example, is described in global coordinates. Compare **geometry space**, **local space**, **device space**.

glyph justification overrides array A style object property used only by layout shapes. It is an array that redefines the justification priorities and behaviors for individual glyphs.

glyph shape A shape type that represents a set of characters or glyphs, each of which is drawn with independent style, location, and orientation.

glyph substitutions array A style object property used only by layout shapes. It is an array specifying substitute glyphs for those that would normally be displayed in a style run.

GraphicsBug A tool for debugging QuickDraw GX applications; its mode of use and command set are analogous to those of MacsBug.

graphics client A reference to a block of memory (the graphics client heap) used for an application's objects.

graphics client heap The part of computer memory in which QuickDraw GX allocates its objects and data structures. Compare **application heap**.

grayscale Consisting entirely of shades of gray.

gray space A color space whose single component is the lightness or brightness of a color. Same as **luminance color space**.

hairline The thinnest possible line that can be drawn on a device.

halftone A QuickDraw GX data structure—also a property of a view port object—that specifies a pattern and a set of colors. A halftone is used to achieve a greater range of colors than may be available on a display device. See also **angle**,

background color, **dot color**, **dot type**, **frequency**, **tint type**.

hierarchy See **view port hierarchy**.

highlight mode A transfer mode type in which the source component and operand component are swapped in the destination. Other component values in the destination are ignored.

hit point In hit-testing, the point (commonly corresponding to a mouse-down location) to be tested for coincidence with a shape or part of a shape.

hit-test info structure A structure, filled out by a hit-testing function, that contains the results of a hit-test.

hit-testing The conversion of a specific geometric location, such as pixel position in a view port, to logical location (part, control point, or glyph) in the geometry of a shape object. Hit-testing is used to highlight or activate parts of geometric shapes or to highlight or draw a caret within the displayed text of a typographic shape.

hit-test parameters A property of a transform object. They consist of a shape-parts mask and a tolerance that together specify the conditions of a hit-test.

HLS color space A color space whose three components measure the hue, lightness, and saturation of a color.

hollow fill See **closed-frame fill**.

HSV color space A color space whose three components measure the hue, saturation, and value (similar to lightness) of a color.

identity mapping A mapping matrix in which all elements are 0 except those along the diagonal, which are 1.0. An identity mapping leaves unchanged whatever it is applied to.

indexed color space A color space whose single component defines an index into a list of color values in a **color set**.

ink A QuickDraw GX object associated with a shape object. An ink object contains information that affects the color of a shape and the transfer mode with which it is drawn.

inverse even-odd fill A shape fill that is the inverse of **even-odd fill**.

inverse fill See **inverse even-odd fill**.

inverse solid fill See **inverse even-odd fill**.

inverse winding fill A shape fill that is the inverse of **winding fill**.

job object A printing object that holds the primary printing information for a document.

join A style object property. It is the appearance (such as rounded or sharp, or any other geometric shape) of corners where a shape's lines or contours meet.

justification The process of adding space or otherwise increasing the spacing of glyphs to align text with both its left and right margins. Justification is a form of alignment, and is incremental; text that completely fills the space between both margins is **fully justified**.

kerning adjustments array A style object property. It is an array specifying changes to the font-specified kerning for pairs of glyphs in a style run. (This property is used by layout shapes only.)

L*a*b* space A universal color space, designed to create perceptually linear gradations between colors, that is a nonlinear transformation of the Munsell color-notation system.

layout shape A shape type that represents a line of text that can be drawn using sophisticated typographic formatting and glyph substitutions.

line shape A shape type that represents a straight line.

load To return an unloaded QuickDraw GX object from external storage to memory. QuickDraw GX automatically and transparently loads and unloads objects in the course of managing memory; an application need never know whether an object it accesses is currently loaded or unloaded.

local coordinates Paired values that specify a size or location in local space.

local space The coordinate system, interior to a view port, resulting from the application of the transform mapping to the geometry of a shape object. Compare **geometry space**, **global space**, **device space**.

lock To prevent an object in the QuickDraw GX heap from being moved. You can lock some QuickDraw GX objects and manipulate their properties directly, instead of using functions to copy them into and out of application memory. See also **unlock**.

luminance color space A color space whose single component is the lightness or brightness of a color. Same as **gray space**.

luminance tint A halftone tint type in which the tint color is specified by the luminance of the input color.

L*u*v* space A universal color space, designed to create perceptually linear gradations between colors, that is a nonlinear transformation of XYZ space.

mapping A 3×3 matrix—a property of a transform object, view port object, and view device object—that specifies the translation, rotation, or distortion to be applied to a shape when it is drawn.

maximum mode A transfer mode type in which the source component replaces the destination component only if the source component has a larger value.

message A command sent by QuickDraw GX to accomplish printing-related tasks.

Message Manager A part of system software, related to QuickDraw GX, that manages **messages**.

method See **dot type**.

migrate mode A transfer mode type in which the destination color component is moved toward the source component by the value of the step specified in the operand component.

minimum mode A transfer mode type in which the source component replaces the destination component only if the source component has a smaller value.

notice A diagnostic message posted by QuickDraw GX when a function is called unnecessarily.

object A private QuickDraw GX data structure. An object has specific **properties** and is accessed through a **reference**.

object sharing The use of the same object by several owners, such as application variables or other objects. Many QuickDraw GX objects can be shared. See also **owner count**, **clone**.

offscreen drawing The process of drawing into an offscreen buffer in preparation for later transfer of the drawn image to the screen.

onscreen view group The view group, created by QuickDraw GX, that includes all view devices for physical display devices.

open-frame fill A shape fill that connects the points of the geometry from start point to end point (but not back to the start point again). Same as **framed fill**.

operand A numerical value used with some transfer mode types (such as blend mode) to affect the outcome of the transfer-mode operation.

OR mode A transfer mode type in which the bits of the source color component and destination color component are combined using an OR operation.

out of gamut Said of a color that cannot be represented on a given device.

over mode A transfer mode type in which the source color is copied to the destination, and the source transparency controls where the destination color shows through.

owner A variable, structure, or QuickDraw GX object that references an object. Many objects can be referenced by more than one variable, and can thus have multiple owners.

owner count A property of some QuickDraw GX objects; it is the number of current references to the object.

paper-type object A printing object that specifies the type and dimensions of the paper printed to.

parent view port A property of a view port object. A view port's parent is that view port immediately above it in the view port hierarchy.

path shape A shape type that represents one or more path contours, each of which is a set of contiguous line segments that can be curved or straight.

pattern A style object property. It is the pattern (actually, any geometric shape, glyph shape, or bitmap shape) to use in filling the geometry of the shape.

pen width A style object property. It is the width of the pen used to draw the shape.

perceptual matching A color-matching method in which all colors produced on the source device are shifted to fit the gamut of the destination device, even those already within the gamut of the destination device. Compare **colorimetric matching**, **saturation matching**.

perspective The altering of a two-dimensional image to give the impression of a third dimension. A mapping can be used to alter the perspective of a shape.

picture shape A shape type that represents a collection of other shapes.

point shape A shape type that represents a single point.

polygon shape A shape type that represents one or more polygon contours, each of which is a set of contiguous straight-line segments.

portable digital document (PDD) A specialized print file that contains all information, including font information, needed to reconstruct and draw the shapes it contains.

post For an error, warning, or notice, to place in an accessible location. QuickDraw GX posts an error, for example, when a function cannot complete successfully.

primitive shape A shape in which the stylistic information is incorporated into the shape's geometry.

printer driver A software module that controls how the contents of a document are spooled, rendered, and sent to a specific output device.

printer object A printing object that represents the capabilities of a physical printer.

print file The spooled version of a QuickDraw GX shape or set of shapes that is the intermediate stage in printing. A print file consists of a stream of flattened QuickDraw GX objects. See also **portable digital document**.

print-file object See **print file**.

printing extension A software module that extends the printing capabilities of QuickDraw GX applications and printer drivers.

printing objects QuickDraw GX objects used for printing. Printing objects include the **job object**, **format object**, **paper-type object**, and others.

priority justification override A style object property used only by layout shapes. It is a structure that redefines the justification priorities and behaviors for whole classes of glyphs.

profile chromaticities A set of color values in a color profile, giving the device-independent colors representing the full intensities of the primary colors on the device.

profile data A property of a color profile object; it consists of a ColorSync color profile structure.

profile response curves A set of curves in a color profile representing the color response of a device as the color intensity ranges from zero to maximum.

property An item or set of data in a QuickDraw GX object. A property of an object is analogous to a field (or member) of a data structure; however, a field is accessed through its name, whereas a property is accessed through a function.

pseudo-Boolean transfer modes Transfer mode types in which the result color is achieved by normalizing the source and destination values and performing simple arithmetic operations whose results are analogous to 1-bit Boolean operations.

QuickDraw GX A sophisticated graphics programming system that is based on objects and provides powerful graphic and typographic capabilities, as well as convenient and flexible printing features.

QuickDraw GX memory The parts of computer memory used by QuickDraw GX, including the **graphics client heap**. Compare **application heap**.

ramp-AND mode A transfer mode type in which the source and destination color components are normalized, and their product (source \times destination) is the result.

ramp-OR mode A transfer mode type in which the source and destination color components are normalized, and the result of (source + destination - source \times destination) is the result.

ramp-XOR mode A transfer mode type in which the source and destination color components are normalized, and the result of (source + destination - 2 \times source \times destination) is the result.

rectangle shape A shape type that represents a single rectangle.

reference A longword value, neither a pointer nor a handle, through which an application accesses a QuickDraw GX object. References are created by QuickDraw GX and passed to applications.

reference white point See **white point**.

remote memory Memory, such as that on an accelerator card, that is not directly addressable. Compare **direct memory**.

result color The color of the destination after drawing has occurred. Compare **source color**, **destination color**.

result color limits In a transfer mode, limits on the permissible values for result color to achieve in transfer-mode calculations. Compare **source color limits**, **destination color limits**.

result matrix A 5×4 matrix, part of the transfer mode structure, that allows you to manipulate the components of the result color after it is calculated.

RGB color space A color space whose three components measure the intensity of red, green, and blue. Used mostly for color video.

rotate To turn about a point. A mapping can be used to rotate a shape about a fixed origin.

run controls A style object property used only by layout shapes. It is a set of values and flags that control various aspects of how the text in a style run is displayed.

run features array A style object property used only by layout shapes. It is an array specifying the set of font features—typographic capabilities as defined by the font—to apply to the text of a style run.

saturation matching A color-matching method in which colors from the source device are shifted to fit the gamut of the destination device in such a way that their saturation (vividness) is preserved. Compare **colorimetric matching**, **perceptual matching**.

scale To proportionally enlarge or shrink. A mapping can be used to scale the geometry of a shape, about a fixed origin, either horizontally or vertically.

shape (1) A graphic or typographic item (such as a geometric shape, a bitmap, or a line of text) created and drawn with QuickDraw GX. (2) A set of QuickDraw GX objects that, taken together, describe the type and characteristics of such a graphic or typographic item. A shape consists of a shape object, a style object, an ink object, and a transform object.

shape cache A cache created and maintained by QuickDraw GX for storing the results of intermediate calculations made prior to drawing a shape.

shape fill A property of a shape object. The shape fill specifies whether and how QuickDraw GX fills in the outlines of a shape that it draws.

shape object A QuickDraw GX object that, along with several other objects, describes a QuickDraw GX shape. A shape object specifies the fundamental type and contents of a shape.

shape part A designation of a part of a shape or its geometry (such as bounding rectangle or corner point) that can be considered in hit-testing. See also **tolerance**.

shape-parts mask For hit-testing, the list of shape parts to be tested against the hit point. See also **tolerance**.

shape type A property of a shape object. The shape type specifies the classification (such as point, line, bitmap, or text) of a particular shape.

sharing See **object sharing**.

skew To progressively distort in a shearing manner. A mapping can be used to skew a shape, about a fixed origin, either horizontally or vertically.

solid fill See **even-odd fill**.

source color The color of a shape or pixel that is to be drawn. Compare **destination color**, **result color**.

source color limits In a transfer mode, limits on the permissible values for source color to use in transfer-mode calculations. Compare **destination color limits**, **result color limits**.

source matrix A 5×4 matrix, part of the transfer mode structure, that allows you to manipulate the components of the source color.

source profile the color profile attached to a color that is to be drawn or converted. The source profile reflects the characteristics of the device on which the color was originally created. Compare **destination profile**.

spool To flatten a QuickDraw GX shape or collection of shapes, and save it to a print file in preparation for printing. Compare **despool**.

spool block A data structure used in conjunction with a **spool function** for flattening and unflattening QuickDraw GX objects.

spool function An application-supplied function that uses a **spool block** to accept flattened data from QuickDraw GX or prepare flattened data for unflattening into objects.

stream format The public format available for describing QuickDraw GX objects. Objects in stream format are considered flattened, and can be interpreted or reconstructed by parsing. Flattened objects are unflattened when they are converted back to object format.

style A QuickDraw GX object associated with a shape object. It contains information that affects the visual appearance of a shape when it is drawn.

synonym A particular kind of tag object, used by QuickDraw GX to provide an alternate representation of an object for printing.

tag list A property of many QuickDraw GX objects. It is an array of references to tag objects associated with the object.

tag object A QuickDraw GX object whose purpose, structure, and content are entirely controlled by the application creating it. Tag objects exist to allow custom information and behavior to be attached to standard QuickDraw GX objects. Tag objects are classified by **tag type**; objects reference their tag objects through a **tag list**.

tag type A longword data type (equivalent to `OSType`) that can be represented by four 1-byte characters, such as 'appl'. Tag types specify the formats of **tag objects**.

text face A style object property. It is the text face—the constructed stylistic variation from plain text—to apply when drawing the text of a shape.

text shape A shape type that represents a line of characters drawn in a single font and style.

text size A style object property. It is the size, in typographic points (72 per inch), to draw the text of a shape.

tint The area ratio of dot color to background color that describes the **tint color** in a halftone.

tint color The actual resultant color produced by a halftone; it is a mixture of the dot color and the background color, in proportions specified by the **tint ratio**.

tint space The color space used by a halftone.

tint type The calculation method, such as luminance tint or color mixture tint, used to determine the **tint color** and the **tint** in a halftone.

tolerance For hit-testing, a value that specifies how close to a **shape part** a hit point must be for the hit-test to be considered successful.

transfer mode A QuickDraw GX data structure—also a property of an ink object—that controls the interaction between the color of a shape and the colors of the background at the location where the shape is drawn.

transfer mode type A specification of the kind of transfer mode—such as copy mode or XOR mode—to apply when drawing a shape or pixel. In QuickDraw GX, same as **component mode**.

transform A QuickDraw GX object associated with a shape object. A transform object contains information that affects the visual appearance of a shape when it is drawn and specifies how the associated shape objects' geometries will be represented in a view port.

translate To move an item. A mapping can be used to translate, or move, a shape by a given amount or to a given location.

tristimulus values The three components of XYZ space, designed to mimic the three kinds of light response of the human retina.

type See **shape type**.

undercolor removal In CMYK color calculation, the removal of some or all of the cyan, magenta, and yellow inks where black ink is to be substituted. See also **black generation**.

unflatten To convert the public, stream-based description of an object or set of objects into the private, native QuickDraw GX object-based format. Compare **flatten**; see also **stream format**.

universal color spaces Color spaces whose colors are device-independent. Universal colors can be compared without the use of color profiles.

unload To move a QuickDraw GX object from memory to temporary external storage. QuickDraw GX automatically and transparently loads and unloads objects in the course of managing memory; an application need never know whether an object it accesses is currently loaded or unloaded.

unlock To free a previously locked object in the QuickDraw GX heap so that it can be moved. See also **lock**.

view device A QuickDraw GX object associated with a view port object. It describes the characteristics of a given physical display device such as a monitor or a printer.

view group A QuickDraw GX object that consists of a grouping of view ports and view devices.

view port A QuickDraw GX object associated with a transform object. It describes the characteristics of the drawing environment for individual QuickDraw GX shapes.

view port hierarchy An ordered arrangement of view ports that allows for such features as windows within windows, including multiple windows within a single window.

view port list A property of a transform object. It is an array of references to the view ports that the shapes associated with that transform can be drawn to.

visible region In a Macintosh window, the part of a window that can be drawn into; defined by the `visRgn` field in the graphics port record. In view ports attached to windows, QuickDraw GX restricts drawing to the window's visible region.

warning A diagnostic message posted by QuickDraw GX when a function completes successfully but may have produced an unexpected result.

white point A specific definition of what is considered white light, represented in terms of Y_{xy} , and usually based on the whitest light that can be generated by a given device. Colors in some color spaces are defined in comparison to a reference white point. See also **Y_{xy} color space**.

winding fill A shape fill that follows the winding-number rule.

XOR mode A transfer mode type in which the bits of the source color component and destination color component are combined using an exclusive-OR operation.

XYZ color space A universal color space whose three components (the **tristimulus values** X, Y, and Z) are means to reflect the fundamental response of the human eye to color.

YIQ color space A universal color space, used for color television transmission, whose components are Y, I, and Q. Y represents luminance and the other two components carry color information.

Y_{xy} color space A universal color space whose three components (the chromaticity coordinates Y, x, and y) are derived from XYZ color space.

zero-length profile A color profile object that contains no profile data. You can specify a zero-length profile in situations in which you do not want color matching to occur.

Index

A

absolute location for a shape 6-24, 6-67
accelerator memory 2-16
add mode
 defined 5-14
 examples of using 5-44, 5-47
 for calculating alpha-channel values 5-24
alignment
 as style object property 3-5
alpha-channel color spaces 4-24
alpha channels 4-24, 5-20
alpha-channel transfer modes 5-20 to 5-25, 5-48 to 5-49. *See also* atop mode, exclude mode, fade mode, over mode
AND mode 5-17
angle
 of a halftone 7-14
 of a halftone on a device 7-83
anti-aliasing 5-24 to 5-25, 5-49
application heap 1-18
arithmetic transfer modes 5-12 to 5-15. *See also* add mode, blend mode, copy mode, migrate mode, maximum mode, minimum mode, no mode
atop mode 5-22
attributes
 as ink object properties. *See* ink attributes
 as shape object property. *See* shape attributes
 as style object property. *See* style attributes, style text attributes
 as view device property. *See* view device attributes
 as view port property. *See* view port attributes
 defined 1-16

B

background color, for a halftone 7-17
base families for color spaces 4-6
bitmaps
 and ink objects 5-11
 as view device property 7-25, 7-26 to 7-27, 7-55, 7-107 to 7-108
 color spaces for 4-23
bitmap shapes 2-10
 defined 1-11
bitmap structure 7-26
black generation 4-14, 4-29

blend mode
 defined 5-14
 examples of using 5-15, 5-44, 5-45, 5-48
Boolean transfer modes 5-16 to 5-18. *See also* AND mode, OR mode, XOR mode

C

caches for shapes 2-16
cap
 as style object property 3-4
child view port list
 as view port property 7-9, 7-18 to 7-19
 functions for 7-86 to 7-87
 setting up 7-46 to 7-47
chromaticities. *See* profile chromaticities
chromaticity 4-16
clamping. *See* pinning
clipping 1-25
clips 1-25
 and drawing 7-30 to 7-39
 and primitive shapes 2-33, 6-7
 as transform object property. *See* transform clip
 as view device property. *See* view device clip
 as view port property. *See* view port clip
cloning objects 1-20 to 1-21. *See also* kinds of under objects
closed-frame fill 2-13
CMProfile structure 4-36
CMYK space 4-14 to 4-15
Collection Manager 1-15
collection objects 1-34
 defined 1-15
 color components 4-6, 4-25
 color-component value 4-25, 4-50
 color conversion 4-26 to 4-30, 4-41 to 4-42, 4-60
 colorimetric matching 4-30
 color limits for transfer modes 5-27 to 5-33, 5-47 to 5-48, 5-54
 destination 5-32, 5-54
 result 5-32 to 5-33, 5-54
 source 5-31, 5-54
color matching 4-26 to 4-32, 4-41 to 4-42, 7-20
 and ColorSync Utilities 4-31, 4-32, 4-42
color-matching methods 4-28, 4-30 to 4-31
color packing 4-6, 4-54

- color profile objects 4-28 to 4-30, 4-35 to 4-38, 4-41 to 4-49, 4-78 to 4-93. *See also* color profiles and ColorSync Utilities 4-30, 4-36 to 4-37, 4-48
 - assigning to colors 4-39
 - constants and data types for 4-57
 - copying, comparing, and cloning 4-44 to 4-45, 4-81 to 4-83
 - creating and disposing of 4-42 to 4-44, 4-79 to 4-81
 - default 4-37
 - defined 1-13, 4-57
 - functions for 4-78 to 4-93
 - loading and unloading 4-45 to 4-46
 - locking and unlocking 4-49, 4-90 to 4-93
 - manipulating profile data in 4-48 to 4-49, 4-88 to 4-93
 - manipulating properties of 4-46 to 4-49, 4-84 to 4-87
 - properties of. *See* color profile properties
 - zero-length profiles 4-37 to 4-38
- color profile properties 4-36 to 4-37, 4-84 to 4-87
 - default values for 4-37
 - owner count 4-36, 4-46, 4-84
 - profile data 4-36, 4-36 to 4-37, 4-48 to 4-49, 4-88 to 4-93
 - tag list 4-36, 4-47, 4-85 to 4-87
- color profiles 4-28. *See also* color profile objects
- colors 4-5 to 4-32, 4-38 to 4-42, 4-57 to 4-61.
 - See also* color profile objects, color spaces, color set objects, color structure
 - as ink object property 5-6, 5-7 to 5-8
 - assigning 4-38 to 4-39
 - color-component value 4-25, 4-50
 - color value 4-25, 4-50 to 4-52
 - comparing and testing 4-40 to 4-41
 - constants and data types for 4-50 to 4-56
 - converting 4-26 to 4-30, 4-31 to 4-32, 4-41 to 4-42, 4-60
 - functions for 4-57 to 4-61, 5-68 to 5-72
 - getting, for a shape on a device 7-119 to 7-120
 - getting and setting 5-42
 - in a color set 4-47 to 4-48, 4-56
 - matching 4-26 to 4-32, 4-41 to 4-42, 7-20
 - out of gamut 4-27, 4-40
- color separations 5-49
- color set objects 4-32 to 4-35, 4-42 to 4-49, 4-62 to 4-77
 - colors in 4-56
 - constants and data types for 4-56 to 4-57
 - copying, comparing, and cloning 4-44 to 4-45, 4-66 to 4-68
 - creating and disposing of 4-42 to 4-44, 4-64 to 4-65
 - default 4-34 to 4-35
 - defined 1-13, 4-56
 - functions for 4-62 to 4-77
 - loading and unloading 4-45 to 4-46
 - manipulating properties of 4-46 to 4-48, 4-69 to 4-73
 - manipulating the colors in 4-47 to 4-48, 4-73 to 4-77
 - properties of. *See* color set properties
- color set properties 4-33 to 4-34
 - color space 4-33
 - color-value array 4-33, 4-34, 4-47 to 4-48, 4-73 to 4-77
 - default values for 4-34 to 4-35
 - owner count 4-33, 4-46, 4-69
 - tag list 4-33, 4-47, 4-70 to 4-73
- color spaces 4-6 to 4-24, 4-55 to 4-56. *See also* colors
 - alpha-channel 4-24
 - as color set property 4-33
 - base families for 4-6
 - CMYK 4-14 to 4-15
 - for bitmaps 4-23
 - for transfer modes 5-25 to 5-27
 - HLS 4-11 to 4-13
 - HSV 4-11 to 4-13
 - indexed 4-22 to 4-23
 - L*a*b* 4-17 to 4-18, 4-18 to 4-20
 - L*u*v* 4-17 to 4-18, 4-18 to 4-20
 - luminance 4-7 to 4-9
 - NTSC 4-20 to 4-22
 - PAL 4-20 to 4-22
 - RGB 4-9 to 4-11
 - XYZ 4-16, 4-18 to 4-20
 - YIQ 4-20 to 4-22
 - Yxy 4-16 to 4-17, 4-18 to 4-20
- color structure 4-26, 4-53, 5-7 to 5-8, 5-51
- ColorSync Utilities
 - and color matching 4-31, 4-32, 4-42
 - and color profiles 4-30, 4-36 to 4-37, 4-48
 - and the default color profile 4-37
- color-value array, as color set property 4-33, 4-34
- color values 4-25, 4-50, 4-52
- Commission Internationale d'Éclairage (CIE) 4-15
- component modes 5-11 to 5-25. *See also* transfer modes
 - alpha-channel 5-20 to 5-25, 5-48 to 5-49
 - atop mode 5-22
 - exclude mode 5-22
 - fade mode 5-22
 - over mode 5-22, 5-48
- arithmetic 5-12 to 5-15
 - add mode. *See* add mode
 - blend mode. *See* blend mode
 - copy mode. *See* copy mode
 - maximum mode 5-14, 5-45, 5-46
 - migrate mode 5-14, 5-44, 5-48
 - minimum mode 5-14, 5-45, 5-46
 - no mode. *See* no mode
- Boolean 5-16 to 5-18
 - AND mode 5-17
 - OR mode 5-17, 5-45, 5-46
 - XOR mode 5-17, 5-45, 5-46
- defined 5-9, 5-11, 5-55
- highlight mode 5-15 to 5-16, 7-13

pseudo-Boolean 5-18 to 5-19
 ramp-AND mode 5-19, 5-45
 ramp-OR mode. *See* ramp-OR mode
 ramp-XOR mode. *See* ramp-XOR mode
 components. *See* color components
 concatenation of mappings 6-26, 7-30, 7-45
 constructive geometry operations
 on transform clips 6-21 to 6-23, 6-48 to 6-53
 conventions and consistencies in programming
 1-41 to 1-44
 coordinates and coordinate spaces 1-28 to 1-32,
 7-31 to 7-39
 device space 1-31 to 1-32, 7-38 to 7-39
 geometry space 1-29, 7-32
 global space 1-30 to 1-31, 7-34 to 7-37
 local space 1-29 to 1-30, 7-33 to 7-34
 copy mode
 and printing 5-50
 as default 5-12
 defined 5-14
 examples of using 5-44, 5-45, 5-46, 5-47
 creating objects 1-9. *See also* kinds of *under* objects
 curve error
 as style object property 3-4
 curve shapes 1-11, 2-9

D

dash
 as style object property 3-4
 debugging 1-39 to 1-40
 debugging version of QuickDraw GX 1-39
 with GraphicsBug 1-40
 deep copying 2-25, 2-58
 default objects 1-17
 desktop printer 1-35
 despooling 1-34
 destination color 4-24, 5-11
 destination color limits 5-32, 5-54
 device angle, of a halftone 7-83
 device matrix 5-8, 5-33 to 5-34
 device space 1-31 to 1-32, 7-38 to 7-39
 measuring a shape in 7-59 to 7-60, 7-116 to 7-118
 dialog boxes, for printing 1-35
 adding panels to 1-36
 printing status dialog box 1-37
 direct memory 2-16
 direct-mode printing 1-37
 disposing of objects 1-9. *See also* kinds of *under* objects
 dither
 ink attributes and 5-9 to 5-10

dithering 5-9, 7-10
 for bitmaps 7-12 to 7-13
 for shapes other than bitmaps 7-11 to 7-12
 dither level. *See* dithers
 dithers
 as view port property 7-8
 characteristics of 7-10 to 7-13
 forced 5-9, 7-12
 functions for 7-80 to 7-81
 manipulating 7-42 to 7-43
 maximum supported level 7-11
 patterns for 7-11
 dot color, for a halftone 7-17
 dot type, for a halftone 7-15 to 7-16, 7-66
 drawing
 and coordinate spaces 1-28 to 1-32, 7-31 to 7-39
 and shape caches 2-16
 basic operation of 1-24 to 1-28, 2-20, 2-35, 7-30
 functions for 2-84 to 2-85
 offscreen 7-29 to 7-30, 7-62 to 7-63
 drivers, printer 1-35

E

empty shapes 1-11, 2-9
 encoding
 as style object property 3-5
 environment (Macintosh). *See* Macintosh environment
 environment (programming). *See* programming
 environment
 error diffusion 7-12. *See also* dithering
 error handling 1-38 to 1-39
 errors
 defined 1-38
 handlers for 1-39
 posting 1-39
 even-odd fill 2-14
 exclude mode 5-22
 exclusive-OR mode. *See* XOR mode
 extensions, printing 1-35

F

fade mode 5-22
 ff macro 2-26
 fill. *See* shape fills
 flatten flags 2-48
 flattening 1-23, 2-22, 2-39 to 2-42
 constants and data types for 2-48 to 2-50
 functions for 2-87 to 2-92

font
 as style object property 3-5
 font objects
 defined 1-14
 font variations
 as style object property 3-5
 format objects
 defined 1-15
 framed fill. *See* open-frame fill
 frequency, of a halftone 7-14 to 7-15
 full shapes 1-11, 2-11

GA–GW

gamuts 4-27
 geometric operations. *See* constructive geometry
 operations
 geometric shapes. *See also* point shapes, line shapes,
 rectangle shapes, curve shapes, polygon shapes,
 path shapes, empty shapes, full shapes
 defined 1-11
 geometry. *See* shape geometry
 geometry space 1-29, 7-32
 global mapping, of a view port 7-79
 global space 1-30 to 1-31, 7-34 to 7-37
 measuring a shape in 7-63 to 7-65, 7-125 to 7-126
 glyph justification overrides array
 as style object property 3-5
 glyph shapes 2-10
 defined 1-11
 local space for 7-34
 glyph substitutions array
 as style object property 3-5
 graphics 1-4
 GraphicsBug 1-40
 graphics client heap 1-18
 graphics client objects
 defined 1-14, 1-38
 graphic shapes. *See also* geometric shapes,
 bitmap shapes, picture shapes
 defined 1-11
 grouping shapes 2-17

GXA

gxAddMode transfer mode 5-14
 gxAllViewDevices view group 7-30
 gxAndMode transfer mode 5-17
 gxAnyNumber constant 1-43
 gxARGB32Space color space 4-10
 gxAtopMode transfer mode 5-22

IN-4

GXB

gxBlendMode transfer mode 5-14

GXC

gxCachedShape shape attribute 2-27
 GXCacheShape function 2-27, 2-62
 GXChangedShape function 2-34 to 2-35, 2-83
 GXCheckColor function 4-40, 4-57
 GXCloneColorProfile function 4-83
 GXCloneColorSet function 4-45, 4-68
 GXCloneInk function 5-59
 GXCloneShape function 2-26, 2-61
 GXCloneStyle function 3-9, 3-13, 3-20
 GXCloneTag function 8-17
 GXCloneTransform function 6-17, 6-37
 gxCMYK32Space color space 4-15
 gxCMYKColor structure 4-50
 gxCMYKSpace color space 4-15
 gxColorIndex structure 4-52
 gxColorPackingTypes enumeration 4-54
 gxColorProfile type 4-57
 gxColorSet type 4-56
 gxColorSpaces enumeration 4-55
 gxColor structure 4-53, 5-51
 gxColorValue1 constant 1-43
 gxColorValue type 4-50
 GXCombineColor function 4-41, 4-59
 gxComponentFlags enumeration 5-55
 gxComponentModes enumeration 5-55
 GXConvertColor function 4-40, 4-41, 4-48, 4-60
 GXCopyDeepToShape function 2-25 to 2-26, 2-58
 gxCopyMode transfer mode 5-14
 GXCopyToColorProfile function 4-81
 GXCopyToColorSet function 4-66
 GXCopyToInk function 5-39, 5-58
 GXCopyToShape function 2-25 to 2-26, 2-57
 GXCopyToStyle function 3-8, 3-18
 GXCopyToTag function 8-15
 GXCopyToTransform function 6-17, 6-35
 GXCopyToDevice function 7-100
 GXCopyToViewPort function 7-44, 7-72

GXD

gxDeviceAttributes enumeration 7-68
 gxDeviceAttribute type 7-68
 GXDifferenceTransform function 6-21 to 6-23, 6-51
 gxDirectShape shape attribute 2-34
 GXDisposeColorProfile function 4-80

GXDisposeColorSet function 4-43, 4-65
 GXDisposeInk function 5-38, 5-57
 GXDisposeShapeCache function 2-27, 2-63
 GXDisposeShape function 2-25, 2-55
 GXDisposeStyle function 3-7, 3-17
 GXDisposeTag function 8-8, 8-14
 GXDisposeTransform function 6-16, 6-34
 GXDisposeViewDevice function 7-53, 7-99
 GXDisposeViewGroup function 7-63, 7-122
 GXDisposeViewPort function 7-41, 7-71
 gxDotTypes enumeration 7-66
 gxDotType type 7-66
 GXDrawShape function 2-35, 2-84

GXE

gxEnableMatchPort attribute 7-20
 GXEqualColorProfile function 4-82
 GXEqualColorSet function 4-67
 GXEqualInk function 5-59
 GXEqualShape function 2-26, 2-60
 GXEqualStyle function 3-9, 3-19
 GXEqualTag function 8-16
 GXEqualTransform function 6-36
 GXEqualViewDevice function 7-101
 GXEqualViewPort function 7-73
 gxExcludeMode transfer mode 5-22
 GXExcludeTransform function 6-21 to 6-23, 6-53

GXF

gxFadeMode transfer mode 5-22
 gxFlattenFlags enumeration 2-48
 gxFlattenFlag type 2-48
 GXFlattenShape function 2-39, 2-88
 gxForceDitherInk attribute 7-12

GXG

GXGetColorDistance function 4-40, 4-58
 GXGetColorProfile function 4-88
 GXGetColorProfileOwners function 4-46, 4-84
 GXGetColorProfileStructure function 4-92
 GXGetColorProfileTags function 4-85
 GXGetColorSet function 4-48, 4-73
 GXGetColorSetOwners function 4-46, 4-69
 GXGetColorSetParts function 4-75
 GXGetColorSetTags function 4-70
 GXGetDefaultColorProfile function 4-78

GXGetDefaultColorSet function 4-62
 GXGetDefaultShape function 2-23, 2-52
 GXGetHalftoneDeviceAngle function 7-83
 GXGetInkAttributes function 5-40, 5-61
 GXGetInkColor function 5-42, 5-68
 GXGetInkOwners function 5-41, 5-64
 GXGetInkTags function 5-41, 5-65
 GXGetInkTransfer function 5-43, 5-72
 GXGetShapeAttributes function 2-29, 2-74
 GXGetShapeCacheSize function 2-27, 2-64
 GXGetShapeClip function 6-45
 GXGetShapeColor function 5-70
 GXGetShapeDeviceArea function 7-118
 GXGetShapeDeviceBounds function 7-59, 7-116
 GXGetShapeDeviceColors function 7-119
 GXGetShapeFill function 2-28, 2-68
 GXGetShapeGlobalBounds function 7-64, 7-125
 GXGetShapeGlobalViewDevices function 7-58, 7-61, 7-115
 GXGetShapeGlobalViewPorts function 7-95
 GXGetShapeHitTest function 6-80
 GXGetShapeInkAttributes function 5-62
 GXGetShapeInk function 2-30, 2-71
 GXGetShapeLocalBounds function 7-51, 7-96
 GXGetShapeMapping function 6-56
 GXGetShapeOwners function 2-32, 2-76
 GXGetShapeSize function 2-25, 2-56
 GXGetShapeStructure function 2-34 to 2-35, 2-82
 GXGetShapeStyle function 2-30 to 2-31, 2-69
 GXGetShapeTags function 2-32, 2-77, 8-10
 GXGetShapeTransfer function 5-74
 GXGetShapeTransform function 2-30, 2-72
 GXGetShapeType function 2-28, 2-32, 2-66
 GXGetShapeViewPorts function 6-75
 GXGetStyleOwners function 3-11, 3-22
 GXGetStyleTags function 3-14, 3-22
 GXGetTag function 8-10, 8-18
 GXGetTagOwners function 8-20
 GXGetTagStructure function 8-23
 GXGetTransformClip function 6-43
 GXGetTransformHitTest function 6-78
 GXGetTransformMapping function 6-54
 GXGetTransformOwners function 6-39
 GXGetTransformTags function 6-20, 6-40
 GXGetTransformViewPorts function 6-29, 6-73
 GXGetViewDeviceAttributes function 7-110
 GXGetViewDeviceBitmap function 7-55, 7-63, 7-107
 GXGetViewDeviceClip function 7-102
 GXGetViewDeviceMapping function 7-57, 7-105
 GXGetViewDeviceTags function 7-112
 GXGetViewDeviceViewGroup function 7-109
 GXGetViewGroupViewDevices function 7-54, 7-124
 GXGetViewGroupViewPorts function 7-44, 7-123
 GXGetViewPortAttributes function 7-89
 GXGetViewPortChildren function 7-86

GXGetViewPortClip **function** 7-45, 7-74
 GXGetViewPortDither **function** 7-61, 7-80
 GXGetViewPortGlobalMapping **function** 7-57, 7-79
 GXGetViewPortHalftone **function** 7-81
 GXGetViewPortMapping **function** 7-45, 7-48, 7-77
 GXGetViewPortParent **function** 7-84
 GXGetViewPortTags **function** 7-91
 GXGetViewPortViewDevices **function** 7-50, 7-94
 GXGetViewPortViewGroup **function** 7-88
 gxGrayAColor **structure** 4-52
 gxGrayASpace **color space** 4-8
 gxGraySpace **color space** 4-8

GXH

gxHalftone **structure** 7-65
 gxHighlightMode **transfer mode** 5-16
 GXHitTestDevice **function** 7-60, 7-120
 gxHitTestInfo **structure** 2-50
 GXHitTestShape **function** 2-38, 2-86
 gxHLS32Space **color space** 4-13
 gxHLSColor **structure** 4-51
 gxHLSSpace **color space** 4-13
 gxHSV32Space **color space** 4-13
 gxHSVColor **structure** 4-51
 gxHSVSpace **color space** 4-13

GXI–GXK

gxIndexedSpace **color space** 4-23
 gxInkAttributes **enumeration** 5-51
 gxInk **type** 5-50
 GXIntersectTransform **function** 6-21 to 6-23, 6-50

GXL

gxLAB32Space **color space** 4-19
 gxLABColor **structure** 4-52
 gxLABSpace **color space** 4-19
 GXLoadInk **function** 5-40
 GXLoadShape **function** 2-27
 GXLoadStyle **function** 3-10
 GXLoadTransform **function** 6-18
 GXLockColorProfile **function** 4-90
 GXLockShape **function** 2-34 to 2-35, 2-80
 GXLockTag **function** 8-21
 gxLUV32Space **color space** 4-19
 gxLUVColor **structure** 4-52
 gxLUVSpace **color space** 4-19

IN-6

GXM

GXMapShape **function** 6-72
 GXMapTransform **function** 6-64
 gxMapTransformShape **attribute** 2-17, 6-25, 6-26
 gxMaximumMode **transfer mode** 5-14
 gxMigrateMode **transfer mode** 5-14
 gxMinimumMode **transfer mode** 5-14
 GXMoveShape **function** 6-66
 GXMoveShapeTo **function** 6-27, 6-67
 GXMoveTransform **function** 6-58
 GXMoveTransformTo **function** 6-24, 6-59

GXN

GXNewColorProfile **function** 4-79
 GXNewColorSet **function** 4-43, 4-64
 GXNewInk **function** 5-38, 5-56
 GXNewShape **function** 2-24, 2-54
 GXNewStyle **function** 3-7, 3-17
 GXNewTag **function** 8-8, 8-13
 GXNewTransform **function** 6-16, 6-33
 GXNewViewDevice **function** 7-53, 7-63, 7-98
 GXNewViewGroup **function** 7-61, 7-63, 7-122
 GXNewViewPort **function** 7-41, 7-47, 7-63, 7-70
 GXNewWindowViewPort **function** 7-40, 7-41
 gxNoAttributes **constant** 1-43
 gxNoMode **transfer mode** 5-14
 gxNTSC32Space **color space** 4-21
 gxNTSCSpace **color space** 4-21

GXO

gxOrMode **transfer mode** 5-17
 gxOverMode **transfer mode** 5-22

GXP, GXQ

gxPAL32Space **color space** 4-21
 gxPALSpace **color space** 4-21
 gxPortAttributes **enumeration** 7-68
 gxPortAttribute **type** 7-68

GXR

gxRampAndMode **transfer mode** 5-19
 gxRampOrMode **transfer mode** 5-19

gxRampXorMode transfer mode 5-19
 GXResetInk function 5-60
 GXResetShape function 2-31, 2-75
 GXResetStyle function 3-11, 3-21
 GXResetTransform function 6-20, 6-38
 GXReverseDifferenceTransform function
 6-21 to 6-23, 6-52
 gxRGB16Space color space 4-10
 gxRGB32Space color space 4-10
 gxRGBAColor structure 4-51
 gxRGBASpace color space 4-10
 gxRGBColor structure 4-50
 gxRGBSpace color space 4-10
 GXRotateShape function 6-27, 6-70
 GXRotateTransform function 6-25, 6-62

GXS

GXScaleShape function 6-26, 6-27, 6-68
 GXScaleTransform function 6-17, 6-25, 6-60
 gxSelectToEnd constant 1-43
 GXSetColorProfile function 4-89
 GXSetColorProfileTags function 4-86
 GXSetColorSet function 4-48, 4-74
 GXSetColorSetParts function 4-76
 GXSetColorSetTags function 4-71
 gxSetColor union 4-56
 GXSetDefaultColorSet function 4-43, 4-63
 GXSetDefaultShape function 2-23, 2-53
 GXSetInkAttributes function 5-40, 5-62
 GXSetInkColor function 5-42, 5-69
 GXSetInkTags function 5-41, 5-66
 GXSetInkTransfer function 5-43, 5-73
 GXSetShapeAttributes function 2-29, 2-74
 GXSetShapeClip function 6-46
 GXSetShapeColor function 5-42, 5-71
 GXSetShapeFill function 2-28, 2-69
 GXSetShapeGeometry function 2-30, 2-67
 GXSetShapeHitTest function 6-30, 6-81
 GXSetShapeInkAttributes function 5-63
 GXSetShapeInk function 2-30, 2-71
 GXSetShapeMapping function 6-57
 GXSetShapeStyle function 2-30, 2-70
 GXSetShapeTags function 2-32, 2-78, 8-8
 GXSetShapeTransfer function 5-43, 5-75
 GXSetShapeTransform function 2-30, 2-73
 GXSetShapeType function 2-32 to 2-33, 2-66
 GXSetShapeViewPorts function 6-76
 GXSetStyleTags function 3-14, 3-24
 GXSetTag function 8-19
 gxSetToNil constant 1-43
 GXSetTransformClip function 6-23, 6-44
 GXSetTransformHitTest function 6-79

GXSetTransformMapping function 6-55
 GXSetTransformTags function 6-20, 6-41
 GXSetTransformViewPorts function 6-29, 6-74
 GXSetViewDeviceAttributes function 7-111
 GXSetViewDeviceBitmap function 7-55, 7-108
 GXSetViewDeviceClip function 7-103
 GXSetViewDeviceMapping function 7-57, 7-106
 GXSetViewDeviceTags function 7-113
 GXSetViewDeviceViewGroup function 7-54, 7-109
 GXSetViewPortAttributes function 7-42, 7-90
 GXSetViewPortChildren function 7-87
 GXSetViewPortClip function 7-46, 7-47, 7-75
 GXSetViewPortDither function 7-42, 7-61, 7-80
 GXSetViewPortHalftone function 7-43, 7-82
 GXSetViewPortMapping function 7-45, 7-47, 7-78
 GXSetViewPortParent function 7-47, 7-84
 GXSetViewPortTags function 7-92
 GXSetViewPortViewGroup function 7-44, 7-88
 gxShapeAttributes enumeration 2-47
 gxShapeAttribute type 2-47
 gxShapeFills enumeration 2-47
 gxShapeFill type 2-47
 gxShapeParts enumeration 6-32
 gxShapePart type 6-32
 gxShape type 2-46
 gxShapeTypes enumeration 2-46
 gxShapeType type 2-46
 GXSkewShape function 6-26, 6-27, 6-71
 GXSkewTransform function 6-25, 6-63
 gxSpoolBlock structure 2-49
 gxSpoolProcPtr type 2-49
 gxStyle type 3-16

GXT

gxTag data type 8-13
 gxTintTypes enumeration 7-67
 gxTintType type 7-67
 gxTransferComponent structure 5-53
 gxTransferFlags enumeration 5-53
 gxTransferMode structure 5-52
 gxTransform type 6-31

GXU

GXUnflattenShape function 2-40 to 2-42, 2-90
 GXUnionTransform function 6-23, 6-49
 GXUnloadInk function 5-40
 GXUnloadShape function 2-27
 GXUnloadStyle function 3-10
 GXUnloadTransform function 6-18

GXUnlockColorProfile function 4-91
 GXUnlockShape function 2-34 to 2-35, 2-81
 GXUnlockTag function 8-22

GXV, GXW

gxViewDevice type 7-68
 gxViewGroup type 7-69
 gxViewPort type 7-65

GXX

gxXorMode transfer mode 5-17
 gxXYZ32Space color space 4-19
 gxXYZColor structure 4-51
 gxXYZSpace color space 4-19

GXY, GXZ

gxYIQ32Space color space 4-21
 gxYIQColor structure 4-52
 gxYIQSpace color space 4-21
 gxYXY32Space color space 4-19
 gxYXYColor structure 4-51
 gxYXYSpace color space 4-19

H

hairlines 6-8, 6-21
 halftones
 angle 7-14
 as view port property 7-8
 background color 7-17
 characteristics of 7-13 to 7-17
 constants and data types for 7-65 to 7-67
 device angle 7-83
 dot color 7-17
 dot type 7-15 to 7-16, 7-66
 frequency 7-14 to 7-15
 functions for 7-81 to 7-83
 ink attributes and 5-10
 manipulating 7-42 to 7-43
 tint and tint color 7-16 to 7-17
 tint space 7-17
 tint types 7-16 to 7-17, 7-67
 halftone structure 7-14, 7-65 to 7-66
 halftoning 5-9, 7-13
 handlers for errors, warnings, or notices 1-39

IN-8

hierarchies of view ports 7-18 to 7-19, 7-21 to 7-23,
 7-46 to 7-47
 highlight transfer mode 5-15 to 5-16, 7-13
 hit point 1-32
 hit-test info structure 2-36, 2-37 to 2-38, 2-50 to 2-51
 hit-testing
 basic operation of 1-32 to 1-34, 2-20 to 2-21,
 2-36 to 2-38
 constants and data types for 2-50 to 2-51
 functions for 2-36, 2-86 to 2-87
 hit-test info structure 2-36, 2-37 to 2-38, 2-50 to 2-51
 of a shape on a device 7-60, 7-120 to 7-121
 parameters for. *See* hit-test parameters
 hit-test parameters 2-36
 as transform object property 6-6, 6-11 to 6-14
 getting and setting 6-77 to 6-81
 setting up 6-14, 6-30 to 6-31
 shape parts 1-32, 2-20 to 2-21
 shape parts mask 2-36 to 2-37, 6-12 to 6-13
 tolerance 1-32, 2-21, 6-13
 HLS space 4-11 to 4-13
 hollow fill. *See* closed-frame fill
 HSV space 4-11 to 4-13
 hue 4-12

I

identity mapping 1-32, 6-10
 ignoring warnings or notices 1-39
 implementation limits 1-43
 indexed color spaces 4-22 to 4-23
 ink attributes
 as ink object property 5-6
 list of 5-9 to 5-10, 5-51
 manipulating 5-40 to 5-41, 5-61 to 5-64
 ink object properties 5-6 to 5-10
 attributes. *See* ink attributes
 color. *See* colors
 default values for 5-10, 5-60
 owner count 5-6, 5-41, 5-64
 tag list 5-6, 5-41, 5-65 to 5-67
 transfer mode. *See* transfer modes, component modes
 ink objects 5-5 to 5-80
 as shape object property 2-8, 2-30 to 2-31, 2-71 to 2-72
 constants and data types for 5-50 to 5-56
 copying, comparing, and cloning 5-39 to 5-40,
 5-58 to 5-60
 creating and disposing of 5-38 to 5-39, 5-56 to 5-57
 default 5-10
 defined 1-12, 5-50
 functions for 5-56 to 5-76
 loading and unloading 5-40
 manipulating properties of 5-40 to 5-41, 5-61 to 5-67

- manipulating the color of 5-42, 5-68 to 5-72
- manipulating the transfer mode of 5-43, 5-72 to 5-76
- properties of. *See* ink object properties
- resetting default properties 5-60
- inverse even-odd fill 2-14
- inverse fill. *See* inverse even-odd fill
- inverse solid fill. *See* inverse even-odd fill
- inverse winding fill 2-14

J

- job objects
 - defined 1-14
- join
 - as style object property 3-4

K

- kerning adjustments array
 - as style object property 3-5

L

- L*a*b* space 4-17 to 4-18, 4-18 to 4-20
- L*u*v* space 4-17 to 4-18, 4-18 to 4-20
- layout shapes 2-10
 - defined 1-11
- lightness, in HLS space 4-12
- line shapes 1-11, 2-9
- local space 1-29 to 1-30, 7-33 to 7-34
 - measuring a shape in 7-51 to 7-52, 7-96 to 7-97
- locking
 - color profiles 4-49, 4-90
 - shapes 2-17, 2-80
 - tag objects 8-11 to 8-12, 8-21
- luminance 4-7, 5-47
- luminance-based color spaces 4-7 to 4-9

M

- Macintosh environment and QuickDraw GX
 - 1-44 to 1-45
- mappings 1-24 to 1-25, 6-10 to 6-11
 - and drawing 7-30 to 7-39
 - as transform object property. *See* transform mapping
 - as view device property. *See* view device mapping

- as view port property. *See* view port mapping
- changing perspective with 6-10
- concatenating 6-26, 7-30, 7-45
- identity 1-32, 6-10
- rotation with 6-10
- scaling with 6-10
- skewing with 6-10
- translation with 6-10
- map-transform shape attribute. *See*
 - gxMapTransformShape attribute
- matrices for transfer modes 5-33 to 5-34, 5-47 to 5-48
- matrices. *See* mappings
- maximum mode 5-14, 5-45, 5-46
- memory
 - and objects 1-18 to 1-23
 - application heap 1-18
 - direct vs. remote (accelerator) 2-16
 - graphics client heap 1-18
 - memory management 1-18 to 1-19, 1-38
 - migrate mode 5-14, 5-44, 5-48
 - minimum mode 5-14, 5-45, 5-46
 - moving a shape. *See* translation operations
 - MySpoolProc application-defined function 2-91

N

- no fill (shape fill) 2-13
- no mode
 - defined 5-14
 - examples of using 5-47, 5-49
 - for calculating alpha-channel values 5-24
- non-debugging version of QuickDraw GX 1-39
- notices
 - defined 1-38
 - handlers for 1-39
 - ignoring 1-39
 - posting 1-39
- NTSC space 4-20 to 4-22

O

- object properties 1-15 to 1-17
 - attributes 1-16
 - default 1-17
 - defined 1-8, 2-6
 - owner count 1-16, 1-20
 - references 1-16
 - tag list 1-17
- object references 1-16, 1-19 to 1-20
 - defined 1-8

- objects 1-7 to 1-49. *See also* collection objects
 - and memory 1-18 to 1-23
 - cloning 1-20 to 1-21
 - creating 1-9
 - default 1-17
 - defined 1-8
 - disposing of 1-9
 - flattening 1-23
 - kinds of. *See* color profile objects, color set objects, font objects, graphics client objects, ink objects, printing objects, shape objects, style objects, tag objects, transform objects, view device objects, view group objects, view port objects
 - loading and unloading 1-21 to 1-22
 - locking and unlocking 1-22
 - properties 1-15 to 1-17
 - references to 1-19 to 1-20
 - sharing 1-19 to 1-20
 - summary diagram of 1-49
 - unflattening 1-23
 - object sharing 1-19 to 1-20
 - offscreen drawing 7-29 to 7-30, 7-62 to 7-63
 - offscreen view groups 7-29 to 7-30, 7-62 to 7-63
 - onscreen view group 7-7, 7-29 to 7-30
 - open-frame fill 2-13
 - operand 5-12
 - OR mode 5-17, 5-45, 5-46
 - out-of-gamut colors 4-27, 4-40
 - over mode 5-22, 5-48
 - owner count 1-20
 - as color profile property 4-36, 4-46, 4-84
 - as color set property 4-33, 4-46, 4-69
 - as ink object property 5-6, 5-41, 5-64
 - as shape object property 2-9, 2-31 to 2-32, 2-76 to 2-77
 - as style object property 3-6, 3-11 to 3-13, 3-22
 - as tag object property 8-4, 8-11, 8-20, 8-21
 - as transform object property 6-7, 6-19 to 6-20, 6-39
 - defined 1-16
- P**
-
- packing, color 4-6, 4-54
 - PAL space 4-20 to 4-22
 - panels, adding to printing dialog boxes 1-36
 - paper-type objects
 - defined 1-15
 - parent view port
 - as view port property 7-8, 7-18 to 7-19
 - functions for 7-84 to 7-85
 - setting up 7-46 to 7-47
 - path shapes 1-11, 2-10
 - pattern
 - as style object property 3-4
 - PDD. *See* portable digital document
 - pen width
 - as style object property 3-4
 - perceptual matching 4-30
 - perspective operations 6-10
 - picture shapes
 - defined 1-11, 2-11
 - local space for 7-34
 - unique items in 2-17
 - pinning, of colors 5-28, 5-32 to 5-33, 5-54
 - point shapes 1-11, 2-9
 - polygon shapes 1-11, 2-10
 - portable digital document (PDD) 1-34, 1-37
 - posting errors, warnings, and notices 1-38
 - primitive shapes 2-33, 6-7
 - printer drivers 1-35
 - printer objects
 - defined 1-15
 - print file objects 1-34
 - print-file objects. *See also* portable digital document
 - defined 1-15
 - print files 1-37
 - printing 1-6, 1-34 to 1-37. *See also* printing objects, printing dialog boxes
 - transfer modes and 5-49 to 5-50
 - printing dialog boxes 1-35
 - adding panels to 1-36
 - status dialog box 1-37
 - printing extensions 1-35
 - printing modes 1-37
 - printing objects. *See also* job objects, format objects, paper-type objects, printer objects, print-file objects
 - defined 1-14 to 1-15
 - printing status dialog box 1-37
 - priority justification override
 - as style object property 3-5
 - profile chromaticities 4-28
 - profile data, as color profile property 4-36, 4-36 to 4-37, 4-48 to 4-49, 4-88 to 4-93
 - profile response curves 4-29
 - programming environment 1-38 to 1-45
 - conventions and consistencies 1-41 to 1-44
 - debugging. *See* debugging
 - error handling 1-38 to 1-39
 - implementation limits 1-43. *See also* Macintosh environment
 - setting up QuickDraw GX memory 1-38
 - properties. *See* object properties
 - pseudo-Boolean transfer modes 5-18 to 5-19.
 - See also* ramp-AND mode, ramp-OR mode, ramp-XOR mode

Q

QuickDraw GX, general features of 1-3 to 1-7
 compatibility with QuickDraw 1-4
 debugging and non-debugging versions 1-39
 graphics 1-4
 limitations to 1-7
 printing 1-6
 programming conventions and consistencies
 1-41 to 1-44
 QuickDraw GX memory 1-18 to 1-19
 relationship to Macintosh environment 1-44 to 1-45
 typography 1-5

R

ramp-AND mode 5-19, 5-45
 ramp-OR mode
 defined 5-19
 examples of using 5-45, 5-46, 5-48
 for calculating alpha-channel values 5-24
 ramp-XOR mode
 defined 5-19
 examples of using 5-45, 5-46
 for calculating alpha-channel values 5-24
 rectangle shapes 1-11, 2-10
 references. *See* object references
 remote memory 2-16
 result color 5-11
 result color limits 5-32 to 5-33, 5-54
 result matrix 5-8, 5-33 to 5-34
 RGB-based color spaces 4-9 to 4-13
 RGB space 4-9 to 4-11
 rotation operations 6-10
 causing change in shape type 6-27
 using shape geometry 6-27 to 6-28, 6-70
 using transform mapping 6-24 to 6-26, 6-62
 run controls
 as style object property 3-5
 run features array
 as style object property 3-5

S

saturation 4-12
 saturation matching 4-30
 scaling operations 6-10
 using shape geometry 6-27 to 6-28, 6-68
 using transform mapping 6-24 to 6-26, 6-60
 scrolling, in a view port 7-19, 7-22 to 7-23, 7-47 to 7-49

shape attributes
 as shape object property 2-8
 list of 2-16 to 2-18, 2-47
 manipulating 2-28 to 2-29, 2-74 to 2-75
 shape caches 2-16
 shape fills
 as shape object property 2-8
 closed-frame fill 2-13
 even-odd fill 2-14
 inverse even-odd fill 2-14
 inverse winding fill 2-14
 list of 2-13 to 2-15, 2-46 to 2-47
 manipulating 2-28 to 2-29, 2-68 to 2-69
 no fill 2-13
 open-frame fill 2-13
 valid shape types for 2-15
 winding fill 2-14
 shape geometry 1-10
 as shape object property 2-8
 contents of 2-11 to 2-13
 copying between shapes 2-29 to 2-30, 2-67 to 2-68
 directly manipulating 2-34 to 2-35, 2-80 to 2-84
 shape object properties 2-7 to 2-9
 attributes. *See* shape attributes
 default values for 2-18 to 2-19, 2-23, 2-31, 2-75 to 2-76
 fill. *See* shape fills
 geometry. *See* shape geometry
 ink reference 2-8, 2-30 to 2-31, 2-71 to 2-72
 owner count 2-9, 2-31 to 2-32, 2-76 to 2-77
 style reference 2-8, 2-30 to 2-31, 2-69 to 2-71
 tag list 2-9, 2-32, 2-77 to 2-79
 transform reference 2-8, 2-30 to 2-31, 2-72 to 2-73
 type. *See* shape types
 shape objects 2-5 to 2-97. *See also* shapes
 absolute location for 6-24, 6-67
 caching 2-27, 2-62 to 2-65
 changing the default 2-23, 2-52 to 2-53
 colors of, on a view device 7-119 to 7-120
 constants and data types for 2-45 to 2-51
 copying, comparing, and cloning 2-25 to 2-26,
 2-57 to 2-62
 copying geometry of 2-29 to 2-30, 2-67 to 2-68
 converting shape type of 2-33, 2-66 to 2-67
 creating and disposing of 2-24 to 2-25, 2-54 to 2-56
 default 2-18 to 2-19
 defined 1-10 to 1-11, 2-46
 directly manipulating geometry of 2-34 to 2-35,
 2-80 to 2-84
 drawing. *See* drawing
 flattening. *See* flattening
 functions for 2-51 to 2-92
 grouping 2-17
 hit-testing on a view device 7-60, 7-120 to 7-121
 hit-testing. *See* hit-testing
 loading and unloading 2-18, 2-27 to 2-28

- shape objects (*continued*)
 - locking and unlocking 2-17, 2-34 to 2-35, 2-80 to 2-84
 - manipulating owner count of 2-31 to 2-32, 2-61 to 2-62, 2-76 to 2-77
 - manipulating properties of 2-19 to 2-20, 2-28 to 2-32, 2-65 to 2-79
 - measuring
 - in device space 7-59 to 7-60, 7-116 to 7-118
 - in global space 7-63 to 7-65, 7-125 to 7-126
 - in local space 7-51 to 7-52, 7-96 to 7-97
 - memory size of 2-25, 2-56 to 2-57
 - primitive 2-33, 6-7
 - printing 1-34 to 1-37
 - properties of. *See* shape object properties
 - resetting to default values 2-31, 2-75 to 2-76
 - saving and restoring. *See* flattening, unflattening
 - transforming. *See* transforming shapes
 - types of. *See* shape types
 - unflattening. *See* unflattening
 - view devices of 7-58, 7-115 to 7-116
 - view ports of 7-50, 7-95
- shape parts, for hit-testing 1-32, 2-20 to 2-21, 2-36 to 2-37, 6-12 to 6-13
- shapes. *See also* shape objects
 - component objects of 2-5 to 2-6
 - defined 2-5
- shape types 1-10
 - as shape object property 2-8
 - bitmap shapes 1-11, 2-10
 - converting between 2-33, 2-66 to 2-67
 - curve shapes 1-11, 2-9
 - empty shapes 1-11, 2-9
 - full shapes 1-11, 2-11
 - geometric 1-11
 - glyph shapes 1-11, 2-10
 - graphic 1-11
 - layout shapes 1-11, 2-10
 - line shapes 1-11, 2-9
 - list of 2-9 to 2-11, 2-46
 - manipulating 2-28 to 2-29, 2-66 to 2-67
 - path shapes 1-11, 2-10
 - picture shapes 1-11, 2-11
 - point shapes 1-11, 2-9
 - polygon shapes 1-11, 2-10
 - rectangle shapes 1-11, 2-10
 - text shapes 1-11, 2-10
 - typographic 1-11
 - valid shape fills for 2-15
- skewing operations 6-10
 - using shape geometry 6-27 to 6-28, 6-71
 - using transform mapping 6-24 to 6-26, 6-63
- solid fill. *See* even-odd fill
- source color 4-24, 5-11
- source color limits 5-31, 5-54
- source matrix 5-8, 5-33 to 5-34
- spaces. *See* coordinates and coordinate spaces, color spaces
- spool block structure 2-49 to 2-50
- spool function, for flattening and unflattening 2-49, 2-91 to 2-92
- spooling 1-34. *See also* spool block structure, spool function
- style attributes. *See also* style text attributes
 - as style object property 3-5
 - manipulating 3-11
- style object properties 3-4 to 3-6, 3-10 to 3-14, 3-21 to 3-25
 - alignment 3-5
 - attributes. *See* style attributes
 - cap 3-4
 - curve error 3-4
 - dash 3-4
 - default values for 3-6 to 3-7, 3-11, 3-21
 - encoding 3-5
 - font 3-5
 - font variations 3-5
 - glyph justification overrides array 3-5
 - glyph substitutions array 3-5
 - join 3-4
 - Kerning adjustments array 3-5
 - owner count 3-6, 3-11 to 3-13, 3-22
 - pattern 3-4
 - pen width 3-4
 - priority justification override 3-5
 - run controls 3-5
 - run features array 3-5
 - tag list 3-6, 3-14, 3-22 to 3-25
 - text attributes. *See* style text attributes
 - text face 3-5
 - text size 3-5
- style objects 3-3 to 3-26
 - as shape object property 2-8, 2-30 to 2-31, 2-69 to 2-71
 - constants and data types for 3-16
 - copying, comparing, and cloning 3-8 to 3-10, 3-18 to 3-20
 - creating and disposing of 3-7 to 3-8, 3-16 to 3-18
 - default 3-6 to 3-7, 3-11, 3-21
 - defined 1-12, 3-16
 - functions for 3-16 to 3-25
 - loading and unloading 3-10
 - manipulating owner count of 3-11 to 3-13, 3-22
 - manipulating properties of 3-10 to 3-14, 3-21 to 3-25
 - properties of. *See* style object properties
 - style text attributes. *See also* style attributes
 - as style object property 3-5
 - manipulating 3-11
 - synonyms 1-37

T

-
- tag contents, as tag object property 8-4
 - tag list 8-3
 - as color profile property 4-36, 4-47, 4-85 to 4-87
 - as color set property 4-33, 4-47, 4-70 to 4-73
 - as ink object property 5-6, 5-41, 5-65 to 5-67
 - as shape object property 2-9, 2-32, 2-77 to 2-79
 - as style object property 3-6, 3-14, 3-22 to 3-25
 - as transform object property 6-7, 6-20, 6-40 to 6-42
 - as view device property 7-25, 7-56, 7-112 to 7-115
 - as view port property 7-9, 7-91 to 7-93
 - defined 1-17, 1-18
 - tag object properties 8-4
 - contents 8-4, 8-10 to 8-11, 8-11 to 8-12, 8-18, 8-20
 - owner count 8-4, 8-11, 8-20, 8-21
 - size 8-4, 8-18, 8-20
 - tag type 8-4, 8-5 to 8-6, 8-10 to 8-11
 - tag objects 8-3 to 8-25
 - attaching to other objects 8-12
 - constants and data types for 8-13
 - copying, comparing, and cloning 8-9, 8-15 to 8-18
 - creating and disposing of 8-8, 8-13 to 8-15
 - defined 1-13, 1-17, 8-13
 - directly manipulating contents of 8-11 to 8-12, 8-21 to 8-24
 - functions for 8-13 to 8-24
 - loading and unloading 8-9
 - locking and unlocking 8-11 to 8-12, 8-21 to 8-24
 - manipulating properties of 8-9 to 8-12, 8-18 to 8-21
 - properties of. *See* tag object properties
 - QuickDraw GX behavior and 8-7
 - uses for 1-17 to 1-18, 8-6 to 8-7
 - tag size, as tag object property 8-4
 - tag types 8-3
 - as tag object property 8-4, 8-5 to 8-6
 - list of 8-5 to 8-6
 - text attributes. *See* style text attributes
 - text face
 - as style object property 3-5
 - text shapes 2-10
 - defined 1-11
 - text size
 - as style object property 3-5
 - tint and tint color, for a halftone 7-16 to 7-17
 - tint space, for a halftone 7-17
 - tint types, for a halftone 7-16 to 7-17, 7-67
 - tolerance, for hit-testing 1-32, 2-21, 6-13
 - transfer component flags 5-35, 5-55 to 5-56
 - transfer component structure 5-8, 5-53 to 5-54
 - transfer mode flags 5-8, 5-35 to 5-36, 5-53
 - transfer modes 5-11 to 5-37, 5-44 to 5-50.
 - See also* transfer mode structure
 - and printing 5-49 to 5-50
 - as ink object property 5-6, 5-8 to 5-9
 - color limits for 5-27 to 5-33, 5-54
 - destination 5-32, 5-54
 - result 5-32 to 5-33, 5-54
 - source 5-31, 5-54
 - color space for 5-25 to 5-27
 - flags 5-34 to 5-36
 - functions for 5-72 to 5-76
 - getting and setting 5-43
 - matrices in 5-33 to 5-34
 - summary of operation 5-36 to 5-37
 - types of. *See* component modes
 - transfer mode structure 5-8 to 5-9, 5-52 to 5-53
 - transform clip
 - as transform object property 6-6
 - characteristics of 6-7 to 6-9
 - constructive geometry operations on 6-21 to 6-23, 6-48 to 6-53
 - functions for 6-43 to 6-53
 - getting and setting 6-20, 6-43 to 6-48
 - transforming shapes. *See also* translation operations, scaling operations, rotation operations, skewing operations, perspective operations
 - by altering shape geometry 2-17, 6-26 to 6-28, 6-65 to 6-73
 - by altering transform mapping 2-17, 6-23 to 6-26, 6-58 to 6-65
 - by applying a mapping to the geometry 6-72
 - transform mapping
 - applying another mapping to 6-64
 - as transform object property 6-6
 - characteristics of 6-10 to 6-11
 - functions for 6-53 to 6-65
 - transform object properties 6-6 to 6-14
 - clip. *See* transform clip
 - default values for 6-14, 6-20, 6-38
 - hit-test parameters. *See* hit-test parameters
 - mapping. *See* transform mapping
 - owner count 6-7, 6-19 to 6-20, 6-39
 - tag list 6-7, 6-20, 6-40 to 6-42
 - view port list. *See* view port list
 - transform objects 6-5 to 6-84
 - as shape object property 2-8, 2-30 to 2-31, 2-72 to 2-73
 - constants and data types for 6-31 to 6-32
 - copying, comparing, and cloning 6-16 to 6-18, 6-35 to 6-38
 - creating and disposing of 6-15 to 6-16, 6-18, 6-33 to 6-35
 - default 6-14
 - defined 1-13, 6-31
 - functions for 6-32 to 6-81
 - loading and unloading 6-18
 - manipulating properties of 6-19 to 6-20, 6-38 to 6-48, 6-54 to 6-58
 - manipulating the clip of 6-48 to 6-53

transform objects (*continued*)
 manipulating the view port list of 6-28 to 6-30,
 6-73 to 6-77
 modifying the mapping of 6-23 to 6-26, 6-58 to 6-65
 properties of. *See* transform object properties
 resetting default properties 6-20, 6-38
 translation operations 6-10
 using shape geometry 6-26 to 6-27, 6-66 to 6-68
 using transform mapping 6-24, 6-58 to 6-60
 tristimulus values 4-16
 type. *See* shape type
 typographic shapes. *See also* text shapes, glyph shapes,
 layout shapes
 defined 1-11
 typography 1-5

U

undercolor removal 4-14, 4-29
 unflattening 1-23, 2-22, 2-39 to 2-42
 constants and data types for 2-48 to 2-50
 functions for 2-87 to 2-92
 unique items in a picture shape 2-17
 universal color spaces 4-15 to 4-22
 unlocking
 color profiles 4-49, 4-91
 shapes 2-17, 2-81
 tag objects 8-11 to 8-12, 8-22

V

value, in HSV space 4-12
 video color spaces 4-20 to 4-22
 view device attributes
 as view device property 7-25
 functions for 7-110 to 7-111
 list of 7-27, 7-68
 manipulating 7-56
 view device clip
 as view device property 7-25, 7-26
 functions for 7-102 to 7-104
 manipulating 7-56 to 7-57
 view device mapping
 as view device property 7-25, 7-26
 functions for 7-105 to 7-106
 manipulating 7-56 to 7-57
 view device objects 7-24 to 7-28, 7-52 to 7-60,
 7-97 to 7-121
 colors of a shape on 7-119 to 7-120
 constants and data types for 7-68

copying and comparing 7-52 to 7-54, 7-100 to 7-102
 creating and disposing of 7-52 to 7-54, 7-98 to 7-99
 default 7-28
 defined 1-13, 1-26, 7-5 to 7-7, 7-68
 functions for 7-97 to 7-121
 halftone angle on 7-83
 hit-testing a shape on 7-60, 7-120 to 7-121
 identifying, for a shape 7-58, 7-115 to 7-116
 identifying, for a view port 7-49 to 7-50, 7-94
 manipulating properties of 7-54 to 7-57,
 7-102 to 7-115
 measuring a shape in device space 7-59 to 7-60,
 7-116 to 7-118
 properties of. *See* view device properties
 view device properties 7-25 to 7-27, 7-54 to 7-57
 attributes. *See* view device attributes
 bitmap 7-25, 7-26 to 7-27, 7-55, 7-107 to 7-108
 clip. *See* view device clip
 default values for 7-28
 mapping. *See* view device mapping
 tag list 7-25, 7-56, 7-112 to 7-115
 view group 7-25, 7-55, 7-109 to 7-110
 view group objects 7-29 to 7-30, 7-60 to 7-65,
 7-121 to 7-126
 as view device property 7-25, 7-55, 7-109 to 7-110
 as view port property 7-9, 7-88 to 7-89
 constants and data types for 7-69
 creating and disposing of 7-61 to 7-62, 7-122 to 7-123
 defined 1-13, 7-5 to 7-7, 7-69
 functions for 7-121 to 7-126
 measuring a shape in global space 7-63 to 7-65,
 7-125 to 7-126
 offscreen 7-29 to 7-30, 7-62 to 7-63
 onscreen 7-7, 7-29 to 7-30
 view devices of 7-62, 7-124 to 7-125
 view ports of 7-62, 7-123 to 7-124
 view port attributes
 as view port property 7-9
 functions for 7-89 to 7-90
 list of 7-20, 7-68
 manipulating 7-42 to 7-43
 view port clip
 as view port property 7-8, 7-9 to 7-10
 functions for 7-74 to 7-76
 manipulating 7-44 to 7-46
 view port list
 as transform object property 6-6, 6-11
 functions for 6-73 to 6-77
 manipulating 6-28 to 6-30
 view port mapping
 as view port property 7-8, 7-9 to 7-10
 functions for 7-77 to 7-79
 manipulating 7-44 to 7-45

I N D E X

view port objects 7-7 to 7-23, 7-40 to 7-52, 7-69 to 7-97
 and windows 7-21 to 7-23
 constants and data types for 7-65 to 7-68
 copying and comparing 7-40 to 7-41, 7-72 to 7-74
 creating and disposing of 7-40 to 7-41, 7-70 to 7-72
 default 7-20 to 7-21
 defined 1-13, 1-26, 7-5 to 7-7, 7-65
 functions for 7-69 to 7-97
 getting the global mapping of 7-79
 halftone angle on a device 7-83
 hierarchies of 7-18 to 7-19, 7-21 to 7-23, 7-46 to 7-47
 identifying, for a shape 7-50, 7-95
 manipulating properties of 7-42 to 7-46, 7-74 to 7-93
 measuring a shape in local space 7-51 to 7-52,
 7-96 to 7-97
 properties of. *See* view port properties
 scrolling support 7-47 to 7-49
 view devices of 7-49 to 7-50, 7-94
view port properties 7-7 to 7-20, 7-42 to 7-46
 attributes. *See* view port attributes
 child view port list. *See* child view port list
 clip. *See* view port clip
 default values for 7-20 to 7-21
 dither. *See* dither
 halftone. *See* halftones
 mapping. *See* view port mapping
 parent view port. *See* parent view port
 tag list 7-9, 7-91 to 7-93
 view group 7-9, 7-88 to 7-89

W

warnings
 defined 1-38
 handlers for 1-39
 ignoring 1-39
 posting 1-39
white point 4-17
winding fill 2-14

X

XOR mode 5-17, 5-45, 5-46
XYZ space 4-16, 4-18 to 4-20

Y

YIQ space 4-20 to 4-22
Yxy space 4-16 to 4-17, 4-18 to 4-20

Z

zero-length profiles 4-37 to 4-38

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter Pro printer. Final page negatives were output directly from text files on an Optrotech SPrint 220 imagesetter. Line art was created using Adobe[™] Illustrator and Adobe Photoshop. PostScript[™], the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino[®] and display type is Helvetica[®]. Bullets are ITC Zapf Dingbats[®]. Some elements, such as program listings, are set in Apple Courier.

WRITERS

David Bice, Gary McCue

DEVELOPMENTAL EDITOR

Sanborn Hodgkins

ILLUSTRATORS

Sandee Karr, Lisa Hymel, Barbara Carey,
Bruce Lee, Mai-Ly Pham

PRODUCTION EDITOR

Lorraine Findlay

PROJECT MANAGER

Trish Eastman

LEAD WRITER

David Bice

LEAD EDITOR

Laurel Rezeau

LEAD ILLUSTRATOR

Ruth Anderson

ART DIRECTOR /COVER DESIGNER

Barbara Smyth

Special thanks to Cary Clark,
Josh Horwich, Chris Yerga

Acknowledgments to

Pete "Luke" Alexander, Tom Dowdy,
Pablo Fernicola, Dave Good,
Dave Hersey, Gary Hillerson,
Wendy Krafft, Marq Laube, Dan Lipton,
Dave Opstad, Diane Patterson,
Rich Pettijohn, Laine Rapin, Mike Reed