Includes all the games in the book and their source code

Sex, Lies, and Video Games

How to write a Macintosh® Arcade Game

Bill Hensler

Sex, Lies, and Video Games

Bill Hensler

Addison-Wesley Developers Press
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

ISBN 0-201-40757-4

A-W Developers Press is a division of Addison-Wesley Publishing Company.

Sponsoring Editor: Keith Wollman
Project Manager: Vicki L. Hochstedler
Cover design: Jean Seal
Set in 11-point Palatino by A & B Typesetting

1 2 3 4 5 6 7 8 9 -MA- 9998979695
First printing, December 1995

Addison-Wesley books are available for bulk purchases by corporations, institutions, and other organizations. For more information please contact the Corporate, Government and Special Sales Department at (800) 238-9682.

Find A-W Developers Press on the World-Wide Web at:
http://www.aw.com.devpress/

Contents

Preface

What This Book Is About

About the title—*Sex, Lies, and Video Games*—I lied about the sex. To try to compensate for this close brush with false advertising, I offer you this joke.

A game programmer is a walking down the street and hears a voice coming from the gutter. He looks down and sees a frog.

"Hi," says the frog. "I'm really a beautiful princess. Pick me up, give me a kiss, and I'll let you gaze upon my beauty for a whole hour."

So the game programmer bends down, picks up the frog, puts it in his pocket, and keeps walking.

"HEY! Wait a minute," says the frog. "Let me out!" So the game programmer pulls the frog out of his pocket. Now the frog says, "If you kiss me I'll turn into a beautiful princess and I'll give you great sex for a week!"

The game programmer puts the frog back into his pocket.

"WHAT? Hey, get me out of here!" He pulls the frog out again.

"If you kiss me I'll turn into a beautiful princess and I'll give you great sex for a whole year!"

Back into the pocket. The game programmer keeps on walking.

"WAIT A MINUTE! Get me out of here! Please, take me out of your pocket," begs the frog.

So the game programmer does, and the frog asks him, "What's the deal here? I promise you a beautiful princess and great sex for a year, yet you keep putting me back into your pocket?"

The game programmer replies, "I'm a game programmer. I don't have time for sex. But a talking frog is WAY COOL!"

Having charmed you with this attempt at humor, this is where I'm supposed to convince you to part with your hard-earned dollars and purchase this book. I suppose just asking you to trust me and purchase the book won't work, so I'll have to try being honest.

Warning: Unblushingly biased statements follow.

In this book you'll be exposed to the techniques of producing games on Macintosh computers. Several chapters deal with producing animation to use in your games. A few more give insight into how to produce sounds for your games. Another chapter demonstrates how to tie together the elements of graphics and sounds. And in the last chapter, a full-blown arcade game is programmed right before your eyes.

After reading this book you will have passed Game Programming 101: Structure and Techniques. You'll have been exposed to many of the techniques used in producing commercial games, and you'll have established a foundation for producing your own games. The goal of this book is to provide the basis for you to go on to produce fun games that I can acquire and play.

Why I Wrote This Book

This book exists because another one didn't. I wanted a book for myself that explained how to apply the hidden powers of the Mac Toolbox to produce games. More specifically, I wanted to program arcade games. Games with captivating animation and objectionable sounds. In vain I searched B. Dalton for a copy of *Inside Mac: Games*, but failed in my quest. I found a few books focusing on game programming for IBMs and their ilk, but not a single one on programming games for the one true computer, the Mac.

Over pizza I lamented this unbelievable situation to a friend; I know she was a friend because she pretended to care about what I was complaining about. Her insightful solution was for me to write a book. Not exactly the advice I was looking for. Her forceful argument was based on the premise that someone needed to write a book on this oh-so-important subject, so why not me.

With encouragement like that I thought, "Gee, that's a good idea. Why don't I write a book on game programming? I spend way too much time sleeping and doing other useless activities. I owe it to my programming brethren to write this book."

Plus, this mysterious voice kept repeating, "If you write it they will code."

With voices in my head and pizza in my stomach, I decided I would write a book that would be an introduction to game programming for the Macintosh. I received lots of encouragement from the other programmers I knew. Statements like "I guess I would buy a copy if I couldn't find anything better." And "Will you give me a raise if I buy a copy?"

I hope you'll find this book as useful as I wanted the book I couldn't find to be. And here's a bit of advice: If your friends suggest that you write a book, get new friends.

Acknowledgments

While writing this book I have intruded on the lives of many of my friends. This is where I try to suck up and earn their forgiveness by apologizing, I hope that seeing their names in print will make up for any transgression I may have made.

Large thanks goes to my wife, Rhonda Thurlow-Hensler, my editor, conscience, coach, and most of all my friend.

A huge indebtedness goes to Jonathan Swift (no relation to the writer). I call him by his middle name, Dean, to avoid any confusion. He loaned me his Atari game collection—for which I am eternally grateful. Dean also owns a full-sized Marble Madness game that he occasionally lets me play. Dean is also responsible for the title; that and the title *Hex, Bugs and Rock & Roll* was already taken.

Recognition—or is it blame—goes to my friend Rachel Rutherford, who talked me into to this whole crazy project and then ran off to Microsoft. All of the Microsoft barbs contained herein are directed at you Rachel, not you Mr. Gates (you never know when I might need a new job up near Redmond, Wash.). In her copious spare time Rachel also whipped up the nifty cover for this book.

The rest of the illustrations in the book were cooked up by the team of Clint and Jessica Spencer. Without them you'd be looking at my stick figure drawings.

I'd like to thank Andy Peterman, for making sure I crossed my technical T's and dotted my programming i's. Thanks Andy.

My final thanks goes to all my editors at Addison-Wesley. First, Dave Clark, who started this whole thing and then ran off to Microsoft Press (are you starting to detect a trend here?). Then Martha Steffen, who dragged me and this book kicking and screaming to the finish line. A dinner and a book in no way covers the debt I owe her. Thanks also to Martha's assistant Kaethin Prizer, who made sure my AOL messages didn't disappear into the great bit bucket in the sky—for which she deserves great thanks. Vicki Hochstedler, who picked up for Martha and actually saw me cross the finish line; I thank her and her copy editing cohorts for suffering through my "School House Rock" inspired grammar ("conjunction junction what's your function?"). I'm sure there are many others at Addison-Wesley I owe a few beers to. So, if I missed you come find me.

Read Me First

As with any piece of software you install these days, this book has a "Read Me First" section. Contained in this Read Me First is the summary of who should read this book, who shouldn't, what you'll need to take full advantage of the code contained in the book, and the general instructions for how to read this book.

Who This Book Is For

This book's intended audience is anyone who is interested in learning how to program games on Macintosh computers. You don't need any previous game programming experience or an extensive back-

ground in graphics or sound programming. It might help, but it isn't necessary. What you will need is a fondness for computer games.

While a game programming background isn't assumed, you must have some experience programming a Macintosh. This book assumes that you are capable of producing a simple Mac program that involves windows, menus, and other Mac interface elements. Here's a quick prerequisites test.

```
InitGraf(&qd.thePort);
InitFonts();
InitWindows();
InitMenus();
TEInit();
InitDialogs(OL);
InitCursor();
FlushEvents(everyEvent, 0);
```

If this code looks like any other ancient incantation you've run across, then this book isn't for you. I suggest you purchase it anyway as a goal to strive for on your path to Mac programming enlightenment, but then again, I have a few shekels to gain if you decide to purchase this volume.

While this book is intended to teach the techniques of game programming, these skills aren't limited to game programming. The core skills of game programming—animation and sound programming—are useful in many other programming disciplines. Multimedia and music programmers can benefit from the assets contained in this book even if they have no interest in writing games—although I find it hard to imagine a programmer who wouldn't want to write a game. Actually, I can imagine a programmer who doesn't want to write games. They're the ones who write those annoying pieces of software that enable computers to call me up in the middle of the morning to ask if I've thought of having my septic tank cleaned.

This book is for you if you want to program computer games, you possess some experience programming Macs, and you have plenty of spare time and no life. It will also help if you own a Mac or can borrow one for an indeterminate length of time.

Who This Book Is Not For

While I'd like to think this book has universal appeal, the reality is there are some readers this book is not intended for.

If you've written your own game with over 9,000 lines of 68000 assembly code in it, then this book is not for you.

This book isn't for anyone who answers every game-oriented question with, "When I was at Atari, we did the . . ."

If you think reading this book will enable you to get a job at Nintendo, Electronic Arts, or any other game-programmer's dream employer, you will be greatly disappointed; and unemployed.

If you think that three weeks after reading this book you'll write the next Doom, Mortal Kombat, or F-18 Hornet, put this book back on the shelf and head over to the reality section.

What's in This Book

Chapter 1: An introduction of arcade games. Their history, origins, and a theory of why they're fun to play.

Chapter 2: How animation works in your brain and in a computer.

Chapter 3: An in-depth discussion of hand-built offscreens on the Mac. What are they, how to build them, and how to use them for animation within your game.

Chapter 4: Building offscreens using GWorlds and how using GWorlds differs from the hand-built offscreens in the previous chapter.

Chapter 5: An introduction to the techniques of sprite blitting—building a sprite system, or engine.

Chapter 6: A discussion of the heart of any arcade game—sprite collisions.

Chapter 7: The beginnings of building a game-programming class library.

Chapter 8: The play field class, which is the central focus of the sprite library.

Chapter 9: Using what you've learned to understand the relationship between sprites and sprite cels so that you end up with sprites leaping and frolicking across the screen.

Chapter 10: An example game that is a take-off of the first video game, Pong.

Chapter 11: An introduction to the second most essential element of any game—sound.

Chapter 12: Putting it all together in a complex game called Digger.

What You Will Need

This book assumes that you either own or have access to the following:

- Color-capable Macintosh
- Color monitor with at least a 12-inch screen
- Symantec's C++ compiler of at least version 7.0
- A few megabytes of free hard-disk space
- Spare time

What You Should Already Know

To get the most out of this book, you should be experienced with the C language, have a familiarity with C++, and have at least the ability to read 68000 assembly language or a real desire to learn. If the last part intimidates you, don't let it. The amount of assembly code in this book is slight and is explained in excruciating detail.

Mac programming experience is also a prerequisite. You should be comfortable with the Mac and most of the major Mac Toolbox managers: QuickDraw, Window Manager, Dialog Manager, Event Manager. These managers will be enough to get you started. If you've read Dave Mark's books (*Learn C on the Macintosh* and *Learn*

C++ on the Macintosh), you will have the required knowledge to take advantage of this book's contents.

You need to be acquainted with Symantec's development environment: the editor, the compilers, and the debugger. Regrettably, this book isn't a substitute for a good tutorial on Symantec's development products. The tutorials provided with the Symantec system are satisfactory preparation for compiling and running the examples in this book.

How to Use This Book

Most people should read this book starting at the beginning and working toward the end. That seems obvious, but I personally have the habit of jumping around in technical books searching for just the right juicy intellectual morsel. You probably don't want to follow my example when you read this book. The best way to read it is to pick a starting point based on your previous experience.

The first chapter has only light reading with no code; it's more like history and design precepts. You could skip this section, but remember, those who skip game history are bound to repeat it. And you don't want to be the second programmer to create a game based on the dancing, smiling, Kool-Aid pitcher—do you?

Honor students who already have experience with color offscreen graphics, `CopyBits`, and the other elements of offscreen animation can skip right to Chapter 3. If you find you missed something, you can always go back and read the previous chapters.

If you're only interested in the sound area of game programming, you could jump right to Chapter 6 and ignore all this superfluous graphics programming. Likewise, if you have no interest in writing games with sounds, you can read all the chapters except six. But remember, games without sounds are like movies without sound. People only watch them for a little while before yelling "Projectionist!" and demanding their money back.

Conventions Used in This Book

Whenever you see a block of text that looks like this:

```
// Useless code
for(i = 0; i < kLoopForABit; i++)
{
      // Nothing of interest here.
}
```

you are looking at a chunk of code of some interest.

All words that are in the `Courier` font are Mac Toolbox names, field, or structure names used in Mac headers files, or references to functions that are developed as part of the book.

> A section that look like this is a sidebar to the main body of text. These sidebars contain auxiliary technical information that you might be interested in. You can skip these sections, but you won't be getting your full money's worth.

All numbers are decimal unless they are preceded by a dollar sign or use C's or assembly hexadecimal notation, like this: $AB3C, 0xFF34.

> You'll find this kind of boxed text whenever I want to provide some game-related information. These morsels range from great moments in game history to games I think you should try.

Solely Responsible

Game programs aren't the only things that go bump in the night; technical books have been known to contain bugs. I've tried to make sure this tome is pest-free, but if you happen to spot one please notify me. I can be reached either through Addison-Wesley or electronically at SLVG1@aol.com.

1

Introduction to Arcade Games

Before rushing into the "how" of Mac arcade game programming, let's spend some time looking into the what, where, when, and why. What are arcade games, and how are they differentiated from other types of computer games? Where and when did these things come into existence? And finally, why the heck do people play these contraptions?

Computer Game Field Guide

There are tens of thousands of computer games spread across hard disks all over the world (some are even legally owned). All of these

games can be placed in one of eight categories: adventure, role playing, simulation, strategy, war games, sports, puzzles, and, finally, arcade games.

Even though this book focuses on arcade games, you should be aware of the other types of games. As blasphemous as it sounds, you might someday wish to write another style of game. Why, I can't understand, but you might. So here's an overview of the various game genres in case you wish to travel down that heretical path.

Adventure: "Anybody Bring a Map?"

"Somewhere nearby is Colossal Cave, where others have found fortunes in treasure and gold, though it is rumored that some who enter are never seen again. The object of the game is for you to explore the cave and return (safely) with as much treasure as possible. The cave is a mysterious and magical place. You will face many puzzling and perilous challenges as you explore."

```
— Hit <RETURN> to continue —
```

If you were in a computer room in the 1960s, a quick press of the return key would have propelled you from the harsh fluorescent-lit world of reality into the magical and puzzling realm of the Colossal Caves. With that same keystroke, you would have started a journey into the fresh new world of adventure games.

The original adventure games, like Willie Crowther's and Dan Woods's classic FORTRAN game Adventure, were strictly text-based. No nifty 3D graphics, no thunderous 44.1kHz stereo sampled sounds, just the machine-gun rattle of the line printer as it spewed forth paragraph after paragraph of descriptive text. From this humble beginning the path was cleared for all adventure games.

All adventure games, whether the original Adventure or today's Myst, are based on what is called interactive fiction. Interactive fiction differs from normal fiction in the way the flow of the

plot line is traversed. In noninteractive fiction, like books and movies, the stories are linear. Boy meets girl. Girl dumps boy. Boy gets a computer. Boy trashes girl's credit report. Interactive fiction is nonlinear. The reader/player controls the flow of events that make up the story.

Interactive fiction is a misnomer; it isn't totally interactive. The player can only alter the plot line in ways the game designer predicted. Truly interactive fiction would allow the user to control every aspect of the story. This would require the game designer to anticipate and program for all possible scenarios, no matter how trivial. The number of branch points in even a simple story would quickly become too large for even a horde of game programmers to manage. The only way to accomplish full interactivity would be to have the game generate the story as it went along, taking into account all of the actions of the player. The technology for this kind of interactive game isn't quite here yet. You'd need a setup like a holodeck from "Star Trek: The Next Generation."

Adventure games circumvent the restrictions of interactive fiction by giving the player a directed goal. This goal is almost always a quest. The quest plot line enables the game designer to easily move the player along within the plot and yet seemingly allows the player to control the story's flow. Interaction within a quest is usually generated by having the player encounter puzzles. Without solving each puzzle, the player cannot accomplish the quest's goal. Not true interactivity, but still fun.

The only thing about adventure games that has changed over the past few decades has been the medium of delivery. Text was the original form of expression. The player responded by typing simple commands: "Take rock." "Eat Food." "Kill Dragon." The complexity of the commands available to the user increased, but the medium of delivery was still text. Eventually, graphic adventure games were created that allowed the player to see without the burden of reams of descriptive text. Still, the player interacted with the game through the use of text commands. Adventure games evolved to the point they are today with games like Myst, fully graphic adventures completely driven by mouse clicks. Not a single command needs to be typed. Is this a loss? I don't know, but I sure miss being

able to tell the game to perform anatomically impossible actions when I can't get past a puzzle.

Role-playing: "What Is an 80-sided Die Called Again?"

Role-playing games center on the player's assuming the identity of a character. The same could be said for adventure games, but role-playing games go into much greater depth. A character in a role-playing game starts the game with predefined skill levels for the character's attributes. These attributes can include strength, intelligence, fighting skills, or spell-casting ability. As you play, the character's attributes grow. And, as in life, your character starts with an affinity for certain characteristics. During play the character's mastery of his or her natural attributes improves faster than other attributes. A sorceress has natural talents for magic, not swordplay. And no matter how swashbuckling your sorceress is, her true talents lie in thaumaturgy.

> While the Mac is role playing game-challenged, it does have a few worth playing. One of my favorites is Westwood Studios' The Legend of Kyrandia. Even with its "DOS port" stigma, it's a pretty good romp through the lands of might and magic.

Simulation: "Watch Your Six!"

Pull a screaming F-18 into a gut-wrenching Immelmann. Drop a Ferrari Formula One into sixth gear and tear down the straightaway at the Monaco Gran Prix. Reload the cannon in an Abrahms M-1 tank before the smoke coverage clears. Run silent, run deep as the commander of a nuclear attack submarine. All of these fantasies and more can be yours for the price of a couple of floppies. Welcome to computer simulations.

A computer simulation, as the name implies, is an attempt to simulate reality with a computer. Simulations allow the players to participate in activities that might otherwise be impossible. The

odds of someone's giving me the keys to a Ferrari are only slightly better than of their giving me the launch codes for a nuclear submarine. Computer simulations are the only way I'm ever going to get next to a Ferrari without hearing it ask me to "Step away from the vehicle." Simulations allow you to play out any Walter Mitty fantasy you might have without risking cash, life, or future incarceration.

Computer simulations originated with the military. The armed forces figured they could save a few million dollars each day by having F-16s crash in a computer rather than into the ground. Less wear and tear on the pilots too. From these million-dollar toys eventually came the technology that enables me to ram my F-18 Hornet into the middle of the Golden Gate Bridge without a scratch, except to my ego.

The majority of computer simulation games have the player driving an expensive piece of machinery, be it a plane, car, submarine, or a carrier task force. With the player at the helm, the simulation attempts to re-create what it would be like to drive the real thing. True simulators are used for training, and their entire emphasis is on realistic accuracy. Game simulator designers have to strive for accuracy but have to balance accuracy with good game play. A game simulation could easily be accurate to the point of boredom.

You either like simulation games or you don't. You either love the idea of a game with a half-inch-thick manual or you don't. You either think the game designer of Microsoft's Flight Simulator is a genius for making sure the instrument-panel light bulb will burn out if left on too long, or you think she's seriously anal-retentive.

Strategy: "How Does the Castle Thing Work Again?"

When you play chess, backgammon, or poker, you are playing what is classified as a strategy game. Strategy games don't have plot lines or characters. What they have is rules. Mastering the rules and their constraints is the goal of strategy games. Well, not the whole goal. Mastering the rules and beating your opponent to a pulp is the whole goal.

What a computer brings to a strategy game is a built-in opponent. Playing poker by yourself isn't enjoyable or profitable. Playing strip poker alone is even worse. A computer supplies you with an instant opponent. You want to play a good game of chess, but everyone in your house starts every game by asking, "How does the horsie move again?" Well, fire up your computer and challenge it to a game. It never forgets the rules and very rarely cheats.

Most strategy games can also be used as on-line tutors. Your chess is a little rusty? Have the computer show you where you make your mistakes while you're playing. Or save the entire game for later playback and study.

Playing Monopoly with a computer is more enjoyable because it can take care of the mundane game elements and let you focus on your strategy (which should be to buy Boardwalk). This housekeeping can be done even if you're playing with other people and not the computer. In that situation the computer is the game master and referee.

Recently computer implementations of strategy games have added little touches of animation and sound effects to the games. Play a game of chess with Interplay's BattleChess, and a short animation is performed anytime a piece is captured. You haven't played chess until you've seen your rook lumber over and swallow your opponent's bishop. While these vignettes don't change the rules of the game, they make losing more enjoyable.

War Games: "Incoming!"

Do you think Patton was overrated and Rommel should have cleaned his clock? Feel you could have been the difference at Waterloo? Think Custer only made one small mistake? With war games you can put your money where your mouth is. Become a general, an admiral, or a lowly private. Fight the great battles of history. Try out "what if" skirmishes. Practice the battle tactics of the next century.

War games, as the name implies, are strategy games played within the framework of warfare. In becoming a paper general you'll have to deal with all the messy details that prevent real war from being any fun. You'll have to manage supplies, roads, morale, ammunition, communications, strategy, and tactics. Not to mention the death of your soldiers if you make an error.

War games existed long before computers. Heck, they've probably been around as long as war. The level of minutiae that is maintained in advanced board war games is staggering. You need a CPA to get through the first skirmish. What a computer brings to war gaming—besides a built-in opponent—is bookkeeping. The computer can keep track of troop movements and supply levels, simulate the weather and its effect on progress, and take care of all of the other junk that distracts from the game. That lets you focus on the tactics and strategy needed to win.

If you've never played a war game before and are slightly intimidated by the history-book-size manuals that come with most computer war games, I suggest you purchase a copy of Chris Crawford's Patton Strikes Back. This re-creation of the Battle of the Bulge is a great entry point for acquiring a taste for war games. It also serves as a great example of game design.

Another ease-of-entry game is Dan Bunten's Command H.Q. Picture Milton Bradley's game Risk on steroids, and you'll have a idea of Command H.Q.

If war games aren't your thing, maybe you'd like to try a peace game instead. Chris Crawford's Balance of Power is just the ticket. In Balance of Power you get the daunting task of preventing global nuclear war. Your only weapons are carefully worded telegrams, covert actions, foreign aid, and brains. Someday I hope Balance of Power is used as an entry test for presidential candidates. If you can't stop a game from blowing up the world, what chance do you have with the real thing?

Like simulations, war games have conflicting goals. They must provide accurate realism while still maintaining a good fun quotient. The balancing of these two elements is what separates a good war game from just another statistics final.

Sports: "Golf on a Computer? Why?"

Pick a sport, any sport, and there is a computer version of it. I've played everything from cliff diving to tiddlywinks (which is more fun on a computer, but then again, how could it be any less fun?). Why would anyone want to play a sport on a computer instead of doing the real thing? Safety, for one. I'd like to box, but I have a close relationship with my frontal lobes. With Electronic Arts' 4D Boxing I can get pummeled round after round and still enunciate all my vowels. Another reason might be hero worship. By booting up Michael Jordan In Flight—which is only out for IBMs, darn!—I can soar and slam just like the Great One.

I classify computer sports games in three categories: arcade, statistical simulation, and combination. An arcade sports game is one where eye–hand coordination is the primary interface to the game. Most sports game on Nintendo and the Sega system fall into this category. The antithesis of the arcade sports game is the statistical simulation. In these games you play strictly through the statistics. How you play is determined by your strategy and has no bearing on your reflexes. But the type I prefer is a combination of the two: a game that provides a statistical basis on which to play but still allows for a miracle moment to occur.

I have one problem with sports games—they aren't really sports. They're not even close. Playing a computer in a game of one-on-one will never replace the real thing. Making two points on the computer does not provide the same sensation of stopping short, popping up, and draining one from long-distance. Moving a joystick to the right does not provide the sweet feel of hitting a baseball right on the money. And don't get me started about computer golf. Why anyone would want to take golf, a complicated way to take a long walk, and make it into a computer game is a mystery to me. Without the excuse of being outside, why would anyone play golf? With the body removed from the game there is no sport, only a game that uses the sport's rules and paraphernalia as a framework. This doesn't mean they're not fun. Good sports games are great fun (except for golf), but they are no substitute for the real thing.

Puzzle: "It's Those Darn Russians' Fault"

A computer puzzle game is the electronic equivalent of a Rubik's Cube. Unlike other computer games, there is no plot or long-term goal. Well, solving the puzzles might turn out to be a long-term commitment. Puzzle games are addictive. The combination of simple rules and great game play make good games hard to shut down. The archetype of the computer puzzle game is Tetris.

> Two of the best puzzle games you can play on the Mac are Psygnosis's Lemmings and Dynamix's The Even More Incredible Machine.
>
> The Even More Incredible Machine provides you with an entire Rube Goldberg toolkit. Build contraptions out of bowling balls, conveyor belts, squirrel cages, magnifying glasses, and even weirder stuff. Schedule at least a week off from your job to work through all the levels.
>
> With Lemmings, you get to control the destiny and job skills of a group of tiny lemmings. Your simple goal is to get your lemmings from one side of the playing field to the other, but whole sets of devious obstacles stand in your way. A word of caution: Don't boot up Lemmings right before an important deadline. You'll have a lot of explaining to do come the next morning.

One of the more interesting aspects of puzzle games is their universal appeal. Young people, old people (those over 25), men, women—it doesn't matter, they all like puzzle games. No other game category has the same broad appeal.

Arcade: "Please Give Me Just One More Quarter"

Question: What the hell is an arcade game? Answer: A game whose original design purpose was to separate you from your money, one quarter at a time. To do this, an arcade game needs to have simple instructions. If the game comes with more than a single page of instructions, it's not an arcade game. If it comes with more than a hundred pages, you can be sure it's a simulation. To take your

money, an arcade game needs to make sure that you eventually (and hopefully quickly) lose the game. To keep the quarters coming in, the game must continue to be challenging.

What are arcade games?

- Short-lived
- Meant to extract money from your pockets
- Develop eye–hand coordination
- Manipulation of limited resources
- Highly addictive
- Creative
- Follow no formula
- Novellas or short stories of computer games
- Tons of animation and sounds

Arcade Game Origins

Because this book is about producing arcade games for the Mac, you need to know a smattering about the "roots of your labor." In any discipline, it is important to understand where the idioms of your culture originated. Luckily, game history is an enjoyable ride—and there won't be any pop quizzes.

The quickie history presented here covers arcade games from their creation in the 1960s through the mid-1980s. I've skipped or ignored almost anything after the mid '80s. Why? Because not one darn machine installed in an arcade during the last decade has added much of historical interest. Most of the games since the mid '80s are derivative at best; the rest just suck.

Get the idea that this might be a revisionist history of arcade games? You're right—with me as your guide and head revisioner. Enough delay. Let's set the dial of the wayback machine for Boston, 1961.

Spacewar

In 1961 at the Massachusetts Institute of Technology, a PDP-1 was unceremoniously uncrated and installed by Digital Equipment Corp. On that day I'm sure no one turned on the juice to this $120,000 piece of iron and said, "All right, now we can make some cool arcade games." But they should have.

One person at MIT knew the true purpose of the existence of the PDP-1: games. That was Steve Russel, whose friends referred to him as "Slug." Which makes me wonder about Steve's view of friendship. Steve (I refuse to call someone I've never met Slug) was one of the legendary hackers that MIT was giving birth to at an alarming rate during the early 1960s. You couldn't throw a stick into a computer room at MIT without hitting some current or future computer genius.

Steve Russel first saw the PDP-1 as a better machine for doing LISP research than the IBM he was currently shackled to. But after seeing a few graphic demos, Steve overcame his burden of trying to accomplish something useful and decided to create the grand-daddy of all arcade games: Spacewar.

Spacewar was inspired by early graphic demos on the PDP-1 mixed with some campy science fiction. The game's goal was to fly around in your spaceship and blow the other player's spaceship to smithereens. A game design that still works today.

The game began simply enough. You flew a dot by toggling the switches on the computer's front panel. While entertaining, a game this did not make. The dot eventually became a triangular ship, and the ship spawned a cylinder-shaped opponent. With an opponent, the need for a weapon generated the code that allowed one player to shoot at the other. When your torpedo (a dot) hit your opponent, you were rewarded with a simulated pyrotechnic explosion (a bunch of dots flew around the screen where the ship used to be).

Steve's game gave the programmers at MIT what they wanted: a cool game to play and access to the source code. Soon revisions of Spacewar spawned gravity wells that could suck your ship in and crush it; solar winds that required a steady hand on the toggle switches for a successful flight; multiple torpedoes you could have

on the screen simultaneously; and hyperspace jumps, which gave you an element of last-chance escape.

Spacewar was not only a source of software inspiration, but a hardware one to boot. Trying to fly your ship through the perils of space by flipping toggle switches was a challenge. Those inspired guys at MIT took up the challenge and whipped up some joysticks. Scores immediately doubled.

Spacewar appeared on DEC computers everywhere. The 27-page assembly program was even used as a diagnostic test at DEC. I would have loved to hear that rationalization to management: "Yep, Bob, the only program that really tests this machine is this game from MIT. It was written by a Slug Somebody."

Lots of games came out for computers in the 1960s, but none that you could call an arcade game. It took almost a decade for the seeds that Steve Russel and Spacewar planted to bear fruit. It might have happened faster, but the seeds were planted in Utah. So let's set the dial of the wayback machine for 1968 and the University of Utah.

Pong and Nolan

In 1968 the University of Utah granted Nolan Bushnell a degree in electrical engineering. Little did they know that Nolan had also graduated with a minor in arcade games, with emphasis on becoming an industry icon. Like thousands of other engineering students, Nolan spent his spare time Spacewarring in the university's computer labs. This experience, combined with the knowledge of how to apply solder cleanly, prepared Nolan for a rosy future as the Obi Wan Kenobi of arcade games.

While Nolan worked for the videotape giant Ampex by day, he was concocting the future at night. Nolan had decided that people would pay real money to play Spacewar. Of course he was right—students had been paying tuition for years to satisfy their Spacewar cravings. All he had to do was make Spacewar affordable to build. While it was a neat game, at $120,000 a pop a PDP-1 was not a thrifty delivery medium. Nolan's plan of attack was to build a sys-

tem dedicated to playing Spacewar. By throwing out all the electronic junk that wasn't used by Spacewar, he figured that he could build them cheaply enough to resell them. The dedicated Spacewarring system was christened Computer Space. Nolan's plan was to have a Computer Space console raking in money right next to every pinball machine in America. With this dream in his pocket, he quit his day job at Ampex.

He went to work for a pinball manufacturer and convinced them to build a few thousand Computer Space games. They were placed right next to the company's pinball machines, just like in Nolan's dream. The trouble was nobody played them.

In theorizing that everyone who played pinball would want to play his Computer Space game, Nolan had forgotten where those pinball machines hang out: bars, pool halls, and bowling alleys—all locations known to serve alcohol to their patrons. Drunken pinball players aren't usually receptive to a new entertainment medium that requires reading a full page of instructions before mastering challenges like acceleration vectors and gravity wells. Computer Space was a flop, actually more like a thud. Nolan quit the pinball company and went off to try again.

He realized that Computer Space wasn't a bad game, it had just been directed at the wrong audience. What he needed to create was an arcade game with training wheels.

Back at his house Nolan created the simplest of all arcade games, Pong. "Avoid missing the ball for high score" were its instructions. Pong, for those of you who somehow missed the 20th century, was played like Ping-Pong. The name didn't come from Ping-Pong; it came from the sonar pulse sound the game emitted when it served. I guess no one told Nolan that a sonar pulse is called a *ping* and not a *pong*. While the name may have been inaccurate, the game play wasn't. The computer served the ball and the players volleyed it back and forth with their paddles. If you missed the ball, your opponent scored a point. Miss enough and you lost. The players moved their paddles by gently spinning the rotary knobs located at the front of the game. The game play could not have been easier. You could hold a beer and a conversation and still play the game.

Pong's first public appearance was in 1972, in a bar in Silicon Valley. In its first week, it raked in $300. It would have been more, but the coin bucket couldn't hold any more quarters. Pong's maiden voyage had proved a success, and an industry was born.

"Have Fun and Make Money"

To raise money to build Pongs, Nolan needed to form a company. And to form a company you need a name. The first attempt at a name was Syzygy. As hard as it is to believe, some other company had decided to use that tongue twister as its name. Sticking with the theme of hard-to-pronounce, esoteric names, Nolan branded his new company Atari. It's etymology trivia time. *Atari* comes from the Japanese game go. In go, atari roughly translates as being in check in chess.

After a name, the next thing Atari needed was cash. Nolan asked Bally-Midway if they would like to invest in this new adventure. Some executive at Bally-Midway told Nolan, "Thanks, but no thanks, and my girl will validate your parking ticket on your way out." I've always wondered what happened to the executive who made that mistake. Do his friends tease him? "Hey Bob, bypass any three-billion-dollar deals today?" In an ironic twist of fate, Nolan Bushnell would later have his own chance at rejecting a few billion dollars.

Through persistence and strength of personality, Nolan gathered enough money to start building and shipping Pong games. Atari couldn't keep up with demand without more cash. The cash was provided through venture capital, and with more cash Atari was able to produce games beyond Pong.

Gran Trak was Atari's and the world's first arcade driving game. Atari continued the driving theme with Le Mans, Monte Carlo, Night Driver, and Sprint (one, two, four, and eight).

The next big hit for Atari was Tank. In Tank you drove your tank around in a maze, looking for and shooting at your opponent's tank. This was a great game. It made a great rumbling tank sound, and you had to drive the tank with two joysticks, each of which moved only up and down. To move your tank forward, you pushed both

sticks forward. For reverse, both sticks were pulled back. To turn your tank left or right, you pushed the sticks in opposite directions.

The Tank game I played was in a bowling alley in Columbus, Ohio. I'd force my Dad to play me in between frames. He always won, but I still loved the game. I think it was the controls. The funky way you had to steer the tank made me feel like Patton. I was a kid, I didn't know that Patton was a general and didn't have to drive his own tank.

While I was playing Tank in Ohio, Steve Jobs was joining the ranks of Atari employees. With help from Steve Wozniak, Jobs designed the Atari game Breakout. Steve and Steve later went on to form a small company of their own. Jobs first approached Nolan Bushnell for the initial funding of Apple, but Nolan didn't see a market for the underpowered little computers. Nolan, like the Bally executive before him, had just passed on a few billion dollars. Smart move, Pong boy.

> It's brush-with-fame time. I met Nolan Bushnell twice. Neither time was very exciting, but it did happen. The first was in his hotel suite, where he was demonstrating a new videotape editing device for his new company, Vent. Vent didn't last the year, but for about an hour I got to hang with the founder of Atari and drink his free sodas.
>
> The second time was when Nolan tried to sell the Vent technology to the company I was working for at the time. I somehow wangled (I begged) my way into the high-level schmoozing party going on in the CEO's office. I was treated like the peon I was until I asked Mr. Bushnell if he could fix my broken Petster. The room went silent. No one else knew that Nolan was also the father of robotic pets, Petsters. They were supposed to run around and act like annoying live pets, but with an off switch. Petsters was one of the long list of companies that Nolan had founded.
>
> Mr. Bushnell declined to extend my Petster's warranty and went back to schmoozing with the people with money. I went back to being an ignored peon.

After passing on the personal-computer thing, Nolan and Atari decided the real money was in the home market, so Pong was squeezed into a box that you could attach to your TV. Now kids

could annoy their parents until they purchased them their very own Pong. That year I convinced my Mom that my Dad wanted one for Christmas.

In 1976 Warner Communications bought Atari. Warner was interested in the home game division and left the arcade division to fend for itself. Which it did, helping make arcade games a six-billion-dollar-a-year industry.

And the Rest Is...

Atari showed the world that people would pay billions of dollars (a quarter at a time) to play video games. From Atari came the rest of the arcade game alumni: Sega/Gremlin, Centuri, Game Plan, Bally-Midway, Cinematronics, Taito, Stern, Rock-Ola, Universal, Williams, Exidy, Pacific Novelty, Nintendo, Game-A-Tron, Venture Line, Gottleib.

Classic Arcade Games

There's no respect for history in the arcade game industry. A classic of yesterday will be gutted for its monitor and coin box without even a moment of silence for what's been lost. With the cannibals of history owning most of the arcades you're likely to patronize, your odds of being able to play a classic are about nil. To try to overcome this injustice, I offer *Sex, Lies, and Video Games*'s List of Classic Arcade Games. This list contains a short summary of each of the game's plots and what I think made the game a classic.

This is my list; these are the games I think established the video game industry. If I've snubbed your favorite classic of the past, then I either forgot about that game or I think it sucked.

Asteroids (Atari)

With its five controls—rotate left, rotate right, thrust, fire, and hyperspace—and its space theme, Asteroids is a direct descendant of Spacewar. Instead of battling another player, you pilot your ship

around an asteroid field. Watch out for the UFOs that are also flying around in the asteroid field and don't take kindly to your unwanted presence.

Each level starts with a few large asteroids that split when shot by you into smaller and more dangerous asteroids. Your goal is to clear each level of all the asteroids without being clobbered by one. Every so often a UFO will appear and try to blast you out of the sky. UFOs come in two flavors: large and slow, with the inability to hit the broad side of a barn; and small and quick, with enhanced shooting skills. The best way to get a high score is to leave only one tiny asteroid and wait for the UFOs to show up. When the UFOs finally appear—and they will—blast away.

Besides being a great game, Asteroids was the first arcade game to let you personalize your high score. Score big enough and you could brag that DOG, GOD, or SEX had come, seen, and conquered.

> If you want to try a blast from the past, several Mac game programmers have created their own tributes to Asteroids. Asterax, by Michael Hanson, and Maelstrom, by Andrew Welch, are both excellent shareware tributes. Space Madness, by High Risk Ventures, is a commercial salute to Asteroids.

Battlezone (Atari)

You are a tank on a barren, alien battlefield of the future. In this future, you are placed in a valley filled with wire-frame squares and pyramids, and you aren't alone. Out to destroy you are enemy tanks, super-tanks, missiles, and landers, each more dangerous than the first. Watch your radar and never stop moving, and you might stay alive.

Battlezone was the first game that provided a first-person view of a playing field. Instead of looking down on the game, you were in the game. That, combined with the same realistic control scheme borrowed from the original Tank game and great sounds, produced one of the first realistic arcade games.

Battlezone boasted other firsts: it was the first 3D first-person game, and it was the first arcade game to enlist in the army. In 1980 the U.S. Army ordered up a few thousand modified Battlezones for training soldiers. The Army couldn't have its soldiers driving around in something called a Battlezone, so they had the name changed to a proper Army-sounding one, MK-60. At $15,000 each, Atari let the Army call them anything they wanted to.

> The closest Mac equivalent is Randy Frank's Mac BZone, a shareware version of Battlezone. A close cousin of Atari's tank classic is Velocity's Spectre, a combination of capture the flag and Battlezone.

Centipede (Atari)

A switch from Atari's space-theme games was Centipede. Occupying the bottom quarter of the screen, you are the defender of the mushroom patch. With your bug zapper you must exterminate spiders, fleas, scorpions, and of course centipedes. The centipedes are the true challenge of this game. They weave their multipart bodies through the mushroom patch, trying to get to you at the bottom of the screen. If a centipede (or any other resident of the garden) touches you you lose a life. Lose 3 lives dig out another quarter.

The fun in Centipede is in how the centipedes split into smaller centipedes when you hit them. You have to hit every section of the centipede. Each section hit turns into a mushroom that can either aid or hinder you, depending on its location.

Created in 1981, Centipede stands out as the first whimsical arcade game. Its bright colors and innovative use of a trackball won over many people who had never played video games before, including my wife.

> The game FireFall, by Inline games, is a commercial Mac descendant of Centipede. I haven't played this game (hint for a free copy), but I hear it maintains the Centipede tradition of being graphically spectacular.

Crazy Climber (Taito)

You are a human fly scaling the outside of a building using suction-cup handles. Get to the top and you move on to a more challenging building. Trying to stop your ascent are anonymous people who open their windows, birds who do some dropping, mad scientists who throw flowerpots at you, and of course Kong himself, who will give you a playful swat if you get too close.

Crazy Climber was controlled through two four-way joysticks. A joystick in each hand, you attempted your ascent.

This game makes the list based on its uniqueness and ability to pull quarters directly from my pocket. No one has attempted a clone of Crazy Climber on any computer that I know of, leaving it the first and only human fly game.

Death Race (Exidy)

Drive around the playing field in a little car and run over any people you find blocking your path. A tombstone, which you must avoid from now on, and a funeral dirge reward a successful hit and run.

This is a sick game, and one I reluctantly admit to liking. I include it in my classic list as a reminder that Mortal Kombat was not the first game to irk parents and enrage members of Congress.

Exidy, the makers of Death Race, tried to justify the maniacal driving by explaining that the player was slaughtering gremlins, not innocent pedestrians. Nobody bought this weak excuse, and public pressure forced Exidy to pull this popular game from the arcades.

With the popularity of splatter games today, I'm sure that someone will revise this classic. Tasteless ideas never die, they just come back as sequels.

Defender/Stargate (Williams)

You zip along in this horizontal scrolling game in a spaceship protecting the citizens below from being kidnapped. Your ship can move up, down, left, and right. The sky is populated with Landers,

Mutants, Baiters, Bombers, Pods, and Swarmers. The Landers attempt to steal your ten citizens on the surface and drag them to the top of the screen. You prevent this by blasting the Landers with your laser. The rest of the enemies provide cover for the Landers, each in its own unique way.

If all the citizens are removed from the surface, the planet will explode, leaving you to fight in empty space. Don't let this happen. After all, you are the defender.

You are provided with a radar screen that shows the entire surface of the planet at once. In Defender, your weapons complement consists of a laser shot and a smart bomb that destroyed any enemies on the screen when activated. Defender's sequel, Stargate, added Inviso, or cloaking, to your arsenal. Both games offered a hyperspace jump as a last resort.

Defender and Stargate were only for the dedicated gamer. With a button for thrust, reverse direction, hyperspace, smart bombs, laser fire, and invisible on Stargate, combined with the joystick that controlled your altitude, mastering either of these games required several pounds of quarters.

Both games are the archetypes of arcade games, offering relentless action. If you breathe, you'll lose. I've seen several attempts at Defender on both the Mac and the IBM. None of them even comes close to the original. A keyboard is not a replacement for dedicated input controls.

Dig Dug (Atari)

Try to score points and vegetables by digging tunnels into the ground. Also in the ground are balloonlike Pookas and fire-breathing Fygars. Your only weapons are an air pump and your wits. With your pump you can inflate the Pookas and Fygars until they pop. Using your wits (and a shovel), you can dig tunnels under rocks, which will fall and squish anyone following you too closely. Be careful digging out the rocks—a little too slow and you'll be crushed.

I included Dig Dug in my list for the sole reason that I like it. It was an enjoyable game to play, with a good balance of action and strategy. You'll see later how much I like this game.

Donkey Kong (Nintendo)

A simian, looking very much like King Kong, kidnaps the fair princess Daisy and climbs to the top of the unfinished structure of a skyscraper. You guide the hero, Mario, up the beams and ladders, avoiding all the junk the big ape hurls at you.

This game is too cute for words. But nothing is too cute about the $200 million the game brought Nintendo. The designer of Donkey Kong, Sigeru Miyamoto, went on to cast Mario as the star of Nintendo's rise to domination of the home video-game industry.

Donkey Kong makes the list because of the first appearance of Mario and because of the legal battles between Nintendo and MCA Universal. MCA Universal felt that Donkey Kong infringed on their trademark gorilla, King Kong, and wanted some of the cash that Donkey Kong had generated. This battle was to be the first of many focusing on who owned the rights to the characters in video games. For an in-depth view of this battle, check out a copy of *Game Over*, by David Scheff.

As of the writing of this book Nintendo has released a new Donkey Kong for the GameBoy portable game system. Beyond the original level Nintendo added about a hundred more.

Food Fight (Atari)

As Charley Chuck, you must cross the playing field and get to an ice cream cone before it melts. Trying to stop you are the four chefs: Oscar, Angelo, Jaques, and Zorba. These chefs rise out of holes in the ground (don't all chefs?) and start chasing Chuck. Spread out on the floor are piles of food. Chuck grabs some of the food whenever he passes over a pile, and with food in hand Chuck starts chucking food at the chefs. Watch out—the chefs are accomplished food flingers as well. That's it. Throw food and avoid chefs and holes in the ground. No hyperspace, no laser, just one button and one joystick.

Besides being one of my favorite games (I own a full-size arcade version), Food Fight had two firsts. It was the first arcade game to use the Motorola 68000 processor. More important, it was the first

game to have instant replay. If you had an incredible run on your way to the ice cream cone, you might be rewarded with a high-speed instant replay of the entire level, just like on "Monday Night Football." People who played Food Fight ignored the high score; their whole goal was to get an instant replay.

The instant replay feature of Food Fight (as far as I know) has never been used in another arcade game. The Electronic Arts computer game One on One—Larry Bird vs. Dr. J. had an instant-replay feature, but not a single arcade game had it. I have no idea why, it's a great feature.

Galaga (Midway)

This sequel to Midway's Galaxian had the player as the pilot of a ship being attacked by buglike enemies. These bug-eyed monsters perform intricate dances as they dive-bomb your ship. Your only defenses are a particle cannon and quick reflexes.

You have a chance to double your firepower if one of the monsters captures your ship with its tractor beam and drags it to the top of the screen. Hit the monster that nabbed your ship and you can have your previous ship dock up with your current ship and provide another particle cannon. Two ships, twice the firepower.

Galaga earns its classic status for its introduction of power-ups. A power-up is where the programmer allows the player to temporarily increase the abilities of his or her character. Power-ups can add a unique twist to an otherwise monotonous game.

Joust/Joust II (Atari)

You are a lance-wielding knight riding a flying ostrich. You defend yourself against evil knights with lances of their own. Besides the evil knights, you have to worry about falling in the lava, being pulled into the lava by something that lives in there, and, of course, those screaming pterodactyls.

Your controls are a button that causes your ostrich to flap its wings and a joystick for directing your ostrich left or right. Tap the button for one flap. Keep tapping gently to keep your ostrich airborne. Tap too hard and you bang into the ceiling; tap too infrequently and your ostrich will plunge into the lava.

Your goal is to ram the other knight with your lance. The victor is whosever lance is the highest. Knock over an evil knight and he will be transformed into an egg. Pick up the eggs for more points. Don't pick up the eggs fast enough and they will hatch into fresh knights. These new knights are meaner and faster than the knights that spawned them.

Joust earns its classic status on originality alone. It was a great game with exceptional graphics. For a Mac version of Joust, download a copy of John Calhoun's Glypha.

Marble Madness (Atari)

How do you describe a game that has the player be a marble (an aggie, I think), with the player racing against a clock or another marble through a maze right out of a Dr. Seuss book? On your journey through the maze you must avoid the marble munchers, steelies, and acid pools.

Your marble moves when you spin a large trackball in the same direction that you want your marble to roll. No buttons. No joysticks. Marble Madness is one of the simplest games ever designed.

This was one of the more inventive games to come out of Atari, but the Seussesque graphics and premise are not the reason I've included Marble Madness. The reason is the sound. Marble Madness is filled with great sounds, from the underlying musical scores for each level, to the wacky sound effects, finishing off with the tribute to Hendrix when you get a high score. If you are ever lucky enough to get a Marble Madness game of your own, do yourself a favor and hook it up to your stereo.

If you've never played Marble Madness, consider it your duty to find one and play it.

Missile Command (Atari)

In Missile Command you get to play out your own nuclear war nightmares. You are the commander of three batteries of antimissile missiles protecting six cities on the California coast. Enemy missiles come streaking in from the top of the screen. Enemy bombers fly through, dropping cruise missiles targeted for your cities. The enemy also has deadly satellites and smart bombs.

Your controls are one large trackball for aiming your defensive missiles and three buttons that choose which base will fire your missile. Playing tip: The center base's missiles travel faster than those of the other bases.

Missile Command is a classic. No imitation will suffice.

Pac-Man/Ms. Pac-Man et al. (Midway)

Wooka. Wooka. Wooka. If you've never played Pac-Man then you're either ten years old or have spent the last few decades on a South Seas island. Pong may have created arcade games, but Pac-Man created a monopoly.

For those who don't know how to play Pac-Man, here is the gist of the game. You control the Pac-Man, a yellow circle with a smile resembling the logo from the "Sonny & Cher" show, with a four-way joystick. The Pac-Man is in a maze filled with dots. The dots are for the Pac-Man to eat. When all the dots are digested, the level is over. Also in the maze are four ghosts who are trying to catch you. In the four corners of the maze are the power pills that will give Pac-Man the energy to eat the ghosts. He only has the energy for a limited amount of time. On each level a bonus item appears for the Pacster to munch on. These bonus items can range from fruits to galaxians and even hardware like bells and keys.

Pac-Man was the first crossover game. Women liked playing Pac-Man as much as men. This was a first in arcade history. Pac-Man was also the first game with reward sequences. Whenever you got past a certain level you were rewarded with a cartoon vignette. People kept playing to see what the next intermission would be.

My favorite part of Pac-Man were the four ghosts. Each had its own name and personality. Blinky, the red ghost, is hard to shake off your tail once he sees you. Pinky, the pink ghost, is faster than Pac-Man. Do not try to outrun Pinky. Weave through the maze instead; Pinky is easily lost. Inky, the light-blue ghost, is afraid of the Pac-Man and might run away if you run right at him. Watch his eyes—they give away the direction Inky is going to head. Clyde, the yellow ghost, is the aggressive one and will charge at Pac-Man. Clyde is also the slowest of the ghosts. Having characters with distinct personality traits added depth to a delightfully simple game. You could not progress beyond the easy levels without understanding your competition.

Pac-Man became the first video game that had merchandising rights. You could get Pac-Man hats, shirts, watches, lunch boxes, sheets, sleeping bags. If it had a flat surface there was somebody hawking it with a Pac-Man printed on it. Pac-Man even inspired a hit song, "Pac-Man Fever."

Pac-Man led to Ms. Pac-Man, the first politically correct video game. You can tell the difference between Pac-Man and Ms. Pac-Man by the red bow she wears in her, um, circle. Ms. Pac-Man was a huge hit. So in that Hollywood tradition of "a sequel is better than thinking up something new," Midway created Baby Pac-Man, Super Pac-Man, and 3D Pac-Man. These all bombed. And deservedly so.

> For a Mac version of Pac-Man, try John Butler's shareware Macman Classic Pro. It's so accurate that all the level patterns I memorized so long ago still work.

Qix (Taito)

One of the more original games ever designed was Qix (pronounced kicks). No cute plumbers. No swarming UFOs. Just colored rectangles and something called a Qix.

As the player you move a point, called the stix, around the perimeter of the playing field with a four-way joystick. With your stix

you attempt to acquire screen real estate by drawing a border around the area you want. Once a rectangle's boundary has been completed, it is filled with a tastefully chosen color. The larger the area you inscribe, the larger your score. You and your stix move on to the next level after you have secured more than 75 percent of the screen. The speed at which your stix moves is controlled by two buttons. Pressing the slow button cuts your stix's movement speed in half, but doubles the value of any areas you surround. The other button doubles your stix's speed and halves the value of any areas you encompass.

Without some adversaries, Qix would be just a color Etch-A-Sketch that costs you a quarter and gives you a hernia when you try to erase the screen. Your main foe to screen dominance is the Qix. The Qix bounces around the screen like an electronic butterfly, trying to thwart your attempts at area acquisition. If the Qix intercepts your stix or any line of the area you are currently trying to enclose, you lose your stix. Lose three and you'll be digging for a new quarter. The best way to avoid the Qix is to wait until it has flitted to another part of the screen and then grab a chunk of screen. Too bad it won't work. Hang around in one spot too long and a fuse will start that retraces your path. Once the fuse burns down to where your stix is loitering, you're a goner. To keep you on your toes, the perimeters of the areas are patrolled by the Sparxs. Don't let a Sparx touch your stix. You won't like it.

Qix earns its place in the list with its combination of incredibly balanced game play, avoidance of overly cute characters, and the fact that it's probably influenced more screen savers than any other video game.

Robotron 2084 (Williams)

The year is 2084 and the world is being overrun by evil robots. You are the only person who can save mankind; well, not all of mankind, but at least Mom, Dad, and Mikey.

You have two eight-way joysticks, one for movement and the other for directing your relentless laser fire. There are no fire buttons, as you are always shooting. Moving and shooting.

On the screen with you and Mom, Dad, and Mikey are the robots. Not a few robots. Not even tens of robots. Try hundreds of robots. Your goal is to destroy all the robots before they can get to and reprogram Mom, Dad, and Mikey. While you're blasting away at the robots, make sure you don't accidentally blow away Mikey. Mom won't approve and will Dad be mad when he gets home.

Robotron, like Defender, was not a game for beginners. Your first game of Robotron would be over before your quarter hit the bottom of the coin box. I think I spent more money on mastering Robotron than my first-year college tuition. That also probably explains my grades that year. The mastery of Robotron required a dedication that bordered on religious fervor and the acquisition of eye–hand coordination that fighter pilots would kill for. The reward for this commitment was as great as the sacrifices: a nine-digit high score. Only true video game wizards could attain this holy grail of arcadedom.

Robotron 2084 secures its spot on sheer adrenaline alone. Robotron was the first arcade game that made it possible to lose weight while playing.

Space Invaders (Taito)

You're the commander of a lunar base that is being attacked by some rather rhythmically inclined aliens. At first you only hear the thumpa, thumpa, thumpa of the aliens' footsteps. Then you see them, the Space Invaders. It's just you, your roving laser cannon, and four bunkers that stand between the aliens and the American way of life. The aliens march slowly down the screen lined up in neat rows and columns as if they are trying out for *A Chorus Line.* The aliens at the bottom of the columns will start pelting your base with missiles and laser fire. You must avoid the incoming and dish out some laser fire of your own. The only drawback to destroying aliens is that it makes the rest of them increase their pace down the screen.

Three buttons control the laser cannon. Two move your cannon right or left, with the third being the required fire button. Your only

defenses are the four bunkers between you and the alien horde. And the bunkers won't last long, as they are quickly chipped away by the alien laser blasts.

While you're tearing into the marching aliens, pay attention to the top of your screen. Every so often a UFO will skitter across. Nail the UFO with a laser blast and you'll get a 300-point bonus.

Space Invaders was a megahit when released. Today its monotonous game play (you could count your shots to determine when the alien would fire) would relegate it to being quickly gutted for its monitor and cabinet. Space Invaders is a classic not for its staying power but for the path it blazed in game history. Here is just a partial list of the games that can trace their roots back to Space Invaders: Galaxians, Galaga, Gorf, Intruder, Centipede, Millipede, Astro-Blaster, KickMan, Missile Command, Phoenix, Pleiades, and of course Deluxe Space Invaders.

Tail Gunner (Cinematronics)

You're flying along at the back end of fast-moving space freighter. As the tail gunner your mission is to prevent the enemy fighters from getting past your laser batteries and attacking the more vulnerable parts of your ship. Let ten ships get by you and it's game-over time.

You target the opposing fighters with your joystick. On screen, your aiming graticule tracks your joystick movements. Move the joystick, lead the target just the right amount, and press the fire stud on the top of the stick. Boom! Instant space junk. If a fighter sneaks past your barrage of fire, press the shield button under your other hand. A shield grid will be thrown up around your ship. Any fighters hitting this barrier will be instantly repelled and momentarily stunned, making them an easy target for your guns. Even with your awesome weaponry, you'll need great reflexes and an affinity with the Force to stop wave after weaving wave of enemy fighters.

Tail Gunner won its place in my list (and an immense number of my quarters) through the smooth, graceful flights of the enemy ships. They dipped, barrel-rolled, spun, and twisted their way

across the screen, always slightly ahead of my constant laser fire. It didn't matter to me that the ships followed prescribed paths. The thrill of lining up a three-ship kill with one laser shot more than made up for the lack of the originality in their flying.

Cinematronics, the maker of Tail Gunner, produced a whole line of innovative and enjoyable games. With smash games like Star Castle, Rip Off, Armor Assault, Warrior, and even a remake of Spacewar called Space Wars (now there's an original name), Cinematronics established a reputation of great designs. Regrettably, the company was a victim of the arcade industry crash of the mid '80s. Now that Cinematronics is long gone, its games are being heralded as the Citizen Kanes of the arcade.

> If you wish to experience a classic on your Mac, give High Risk Ventures' shareware homage to Cinematronics' Star Castle, Cyclone, a run. Make sure you set Cyclone's preferences to emulate the original game. Now, if only some kind soul would code up a remake of Tail Gunner.

Tempest (Atari)

Atari's Tempest can best be described as Euclid meets the Space Invaders. You flit around the edge of a geometric tube, blasting the various linear enemies into vector dust. The enemies' whole reason for existence is to climb to the top of the tube and haul you down to the bottom to your doom. Clear the tube of all the enemies and you'll get to zoom through the tube on your way to a new geometric shape on the next level. While you're zooming through the tube, make sure you avoid any leftover debris. Failure to do so will ruin your day and your ascent to the next level.

Your interface for playing Tempest is one spinning dial and two buttons. The player's proxy, which resembles a yellow, bent staple, is controlled by spinning the dial. Spinning the dial moves your staple around the edge of the tube. With the first button, you control the blaster fire of your commando staple. By tapping the fire button like a rat slamming a lever in a Skinner box, you can set up a blaz-

ing, deadly line of fire. Combine rapid spinning of the dial with a furious firing of your blaster and you can quickly clear the inside of the tube. When the "spin & shoot" technique isn't enough and you are about to be overrun, slam your hand on the last button, the Super Zapper. One touch of the Super Zapper button and all the Tempest enemies currently climbing the tube are zapped into pixel heaven. You only get one zapper recharge per level, so be frugal in your zapping.

The challenge of Tempest lies in the various enemies that are thrown at you. Each of them has its own method of attack, and all are despicable. Bad as each enemy is individually, it's the combination of their attack methods that will have you heading to the change machine again and again.

On the first level you'll be introduced to the Flippers. A Flipper is red and moves by flip-flopping its way to the top. A Flipper is harmless until it reaches the top, where it becomes lethal. When a Flipper flops on top of you, you will be dragged kicking and screaming to your ruin.

A Tanker is the troop carrier in Tempest. Shoot a tanker and surprise, you get two irritated Flippers. If a Tanker actually makes it to the top of the tube, it will then split into two Flippers that will hunt you down like you owe them money.

While you're playing Tempest you'll see green spirals that move up and down the tube, leaving green spikes in their wake. The beasties that spins out these spikes are called Spikers (of course). A Spiker cannot do any harm by itself; it's the spikes it creates that are your problem. A spike can be slowly chopped down with repeated blaster fire. But while a spike is in the tube it provides an expressway to the top for any Flippers that wish to munch on you. You can't shoot the Flippers riding to the top until you chop down the spike providing the lift. And while you're chopping down that spike you can be sure a Spiker is planting another. Any spikes left lying around at the end of the level become a hazard to your health as you go zooming through the tube on to the next level.

On higher levels, you are introduced to Pulsars. These aren't just neutron stars spewing radio waves; these Pulsars mean business. When a Pulsar is pulsating, the adjacent sections have the

volts and amps of a subway's third rail. And remember, it's the amps that kill you, not the volts.

When, or more probably, if you get beyond the 11th level, you get to do battle with the FuseBalls. Don't be deceived by its innocent, electrified-hairball appearance. A FuseBall's whole reason for existence is to stop you from getting to the 12th level. They lurk around the tube, and when you aren't looking they zip up at light speed and slam into your staple.

Tempest is one of the most lusted after arcade games. Owning a full-size standup version grants you major bragging rights around any group of arcade game aficionados. But be prepared for the jealousy backlash.

> If you aren't lucky enough to have inherited a Tempest machine, you can get a taste by playing Juri Munkkis's Arashi. Or if you're brave enough to own an Atari Jaguar, or know someone who is, check out Atari's rehashing of the classic: *Tempest 2000*. However, after playing either clone you'll soon discover that the spinning dial was a necessity, not an option.

Closing Ceremonies

That's the end of the list. The games were listed in alphabetical order, ending with *T*. I wish I knew of an arcade game that began with the letter *Z*. Hold on. Zaxxon. There, I knew I would think of one. How could I forget the first orthographically scrolling game? Too late for this list, but who knows about next time.

After reading my list you might get the idea that I haven't stepped into an arcade since the Rolling Stones' first farewell tour. You couldn't be further from the truth. I have dropped some serious coinage into some of today's more popular games. I've mastered all of Ryu's moves and can work through all the fatalities in Mortal Kombat (I'm working on my Mortal Kombat II moves, but it's hard when you're snubbed by all the 12-year-olds in the neighborhood). It's not that these games aren't successful. They are, obscenely so. Mortal Kombat alone has brought in over a billion

dollars. That's billion with a *B*. But money and popularity don't make a classic. If that were true, Arnold Schwarzenegger would have a shelf full of Oscars. The inverse is also true; otherwise Cinematronics would still be around producing great games.

What Makes a Game Fun?

Why do people drop good money on these contraptions? What makes a sane man spend hundreds of dollars for the latest 3DO hardware and then try to justify it as research? Psychology. Psychology is what makes a game fun to play. Your brain is what tells your hand you can go on to the next level of Super Mario Land even though your hands and thumbs have been urging you to quit for the last hour. Now, of course, you would like your game creations to provoke the same next-day wrist-numbing pain that all good arcade games provide. The secret to this is a total understanding of how your player's brain works. Well, at least a little insight into what's going on among those frontal lobes while a player is blasting away enemy aliens. Keep in mind that this section is not a substitute for a degree in cognitive psychology, but then again, what is?

Reinforcement

You're a rat in a small cage with just a water bottle and one of those spinning-wheel things that hamsters love and rats don't. Color you one bored rat. Then one day B. F. Skinner, looking particularly dapper in his starched white lab coat, drills a hole in your cage and above the hole installs a lever. Of course, you're a rat so you don't know it's a lever. You think it's a stick. As a rat your first instinct is to eat the stick. Lucky for ol' B. F., the stick is inedible. In your attempt to consume the lever you inadvertently pull it down. A small *swoosh* announces the arrival of a Purina Rat Chow pellet as it slides out of the hole. Being a rat, you aren't surprised at this; food mysteriously appears all the time. After chowing down on that nugget of balanced nutrition, you go back to attacking the stick. Once again the lever is pulled down and once again a scrumptious morsel ap-

pears. Sixteen hours and 354 pellets later, your rodent-size brain spots a connection. If you attack the stick you get food. Cool. This is as close as a rat can get to Nirvana. B. F. Skinner and all future rat torturers would refer to this particular rat heaven as reinforcement.

Reinforcement is the psychological term for getting a subject, be it a rat or a 16-year-old, to do your bidding; this is accomplished by providing a reward for a desired behavior. *A behavior that is positively reinforced by a reward will likely be repeated.* As with most ideas in psychology, this seems obvious to any person with an ounce of common sense. But by giving it a name with at least three syllables, you get to talk about it like it was a great insight into the human psyche.

But how does this tie in with video games? Easy. In your video games you want a certain behavior—playing the game—to be reinforced so that it will continue. So, following in B. F.'s size nine footsteps, you want your game to provide rewards for playing it. What kind of rewards? If your game could spit out a slice of pizza, a couple twenties, and give me a vigorous back massage after I've completed a particularly arduous level, it would be a game I'd continue playing. I'd have no reason ever to leave the house.

In lieu of this, your game can provide the classical rewards of video games: high scores, new levels, fresh puzzles, cute animation, zany sounds, and a whole litany of others. The main point is that your game has to provide some form of reward for playing it, or no one will. The trick is to find the rewards that your players will find enjoyable. Unlike rats, whose reward needs are easily predicted, humans are slightly trickier. For every player that is reinforced by your designed rewards, there is some number of others who will rate playing your game right up there with hernia surgery. Your job is to find the proper balance of rewards that minimizes the hernia flashbacks and promotes future game playing.

Schedules of Reinforcement

Providing rewards in your creation is not enough to insure a successful game. You must dole out your rewards at the right time. Again, psychologists have coined a phrase for the rate at which rewards are presented: schedules of reinforcement.

The easiest form of scheduling is the one B. F. used in his famous experiment. It's called continuous reinforcement. Every time the rat repeats the behavior, a reward is delivered. Now, the interesting part is that the pattern does not have to be continuous to keep that rat interested. If the pellets appear on a random schedule, say, at least once in every ten pulls, the rat will still keep slamming on that lever like a slot machine junkie down to his last quarter. And by avoiding continuous reinforcement, practicing what's referred to as partial reinforcement, B. F. can cut down greatly on his rat chow bill.

What's true for rats, as anyone who has been to Vegas can tell you, is also true for humans. Reinforcement every so often is enough to keep people playing. And interestingly enough, certain behaviors are more strongly reinforced by occasional rather than continuous rewards. In fact, continuous reinforcement can sometimes be detrimental. Slot machines are obviously a bad example here. The best is an old episode of "Family Affair." Buffy and Jody are given everything they desire. You name it—Hot Wheels, candy, ice cream, horsy rides on Mr. French's back, Misses Beasleys—are handed out like tapwater. At the end of 23 minutes Buffy and Jody learn that continuous reinforcement is boring. And that Brian Keith is an idiot. That was a bad example. But it won me one-third of a bet that I could fit in three '60s sitcoms as game-programming examples. Pay attention. Working in "I Dream of Jeannie" and "My Favorite Martian" is going to take deft avoidance of actual information, and I don't want you to miss it.

Enough of this, back to the subject at hand. A better example is food. At one point in my life I pretty much lived on any nutrition that could be extracted from the contents of vending machines. At first I thought my substitution of riboflavin and artificial colors for the basic four food groups was the correct dietary direction for a future programming career. I liked the reinforcement that I was giving my ego, even if Cheetos fingerprints on all my books did reduce what I could get for them at the end of the semester. Fairly quickly, the reward of this continuous reinforcement of proving myself as a worthy programmer through the consuming of meals that can be purchased with loose change died out. Something I liked was killed

by continuous reinforcement. And only through a strict therapy of continental French cuisine and fine wines was I able once again to enjoy the sharp tang and crispy crunch of a handful of Cheetos. Proving that partial reinforcement is better in the long run.

This applies to video games as well. A game that hands out rewards continuously will be quickly discarded for a more challenging one. And a game that doesn't provide any rewards would be boring. Even Pong provided a score.

Extinction

You've been playing the lottery religiously for months. Every week you bet one dollar, and you always win a hundred (I want your system). Then one week you stop winning. You'd probably keep playing for a few more weeks. But you keep losing, so eventually you go cold turkey and quit playing the lottery. You have just suffered through extinction. And it didn't take a giant meteor slamming into the Earth. Your continuous reinforcement was cut off and you decided that it would never come back, so you stopped playing. How long it takes for you to decide to stop playing without reinforcement is called the extinction period.

Now, if you had been playing the same fantasy lottery game for the same number of months and you had only won a single C-note, you would probably keep playing. You won once, so it must be possible for you to win again. And since you can't predict how you won your past windfall, you can't determine when you'll win the next. Partial reinforcement has kept you from ceasing your behavior, extending the extinction period. And it keeps those jobs manufacturing numbered Ping-Pong balls in this country, where they belong.

The trick is to find the right combination of rewards and the proper amount of time between them. Too long, and the user might think that another reward is never going to come. Too short, and the game seems too easy.

The time between rewards might even vary as the game progresses. Good games provide a bevy of rewards at the beginning to encourage the new player to keep playing, and then as the player's

skills increase, the rewards are separated by larger extinction periods. As the time between rewards stretches out, the value of the reward should increase proportionally. It's one those inverse ratio things, like gravity. And like gravity, if you don't have proper balance things can start sucking.

Magnitude of Reinforcement

While the rewards that a game produces need to be spread out over time and be unpredictable in their arrival, the size of the rewards also needs to vary. You can't go giving out free games for every simple task accomplished during a game. Little deeds deserve little rewards. But if you only give out small rewards, your player will eventually cease playing. So you need to scale the rewards to match your game's challenges.

If you were offered a buck to transcribe *Inside Macintosh: AOCE*, all gazillion pages of it, you'd probably mention something about a life to live as you wondered off to the arcade. But if I offered you $100,000 for the same daunting task, your only comment would be, "Single or double spaced?" as you fired up your copy of Word 6. Welcome to *magnitude of reinforcement*, the psychological term that means by varying the size of the reward you can directly alter behavior. It also implies the inverse—if the rewards are too paltry for the task, the task will be ignored. You can't ask a player in an adventure game to go off and save the kingdom with only a thank-you and nice atta-boy. There has to be something worth fighting for, gold, love, a free T-shirt. Something important.

Welcome to the second round of "Jeopardy." The categories are: Words That Rhyme with Munger, Bill Gates's Past Love Interests, Foods Found in Vending Machines, Things Only Steve Jobs Could Say, Famous Apple Code Names, and Macworld Parties of the Past. You're behind by around $2,000. So on the next round do you go for the $500 question or sit by and be beaten by the player who looks amazingly like Andy Hertzfeld? You bet the $2,000, of course. The gain is much larger than the risk. And anyway, it's Alex Trebek's money, not yours. Now if you're in the lead by, oh say, a million

bucks (these are your dream categories, remember), do you risk doubling your winnings and possibly losing it all in the final Jeopardy round? No way, a million bucks in the hand is a heck of a lot better than the home version of "Jeopardy." The point being that at a certain level, players will stick with a sure thing instead of risking their hard-won high scores, especially when the difference between $1,000,000 and $10,000,000 is psychologically minimal. They're both just really large numbers.

Delay of Reinforcement

Any behavior that is followed by reinforcement will happen more often. The shorter the delay between the behavior and the reinforcement, the faster the behavior will increase in frequency. Short periods between reinforcement amplify the reinforcement.

Any behavior that is not quickly reinforced will be killed by a behavior that is. Most New Year's resolutions fall into this category. A good example that proves this principle is my last New Year's resolution. I decided that starting the new year I would try to reduce my ever-expanding girth. Each day I resolved that during particularly long compiles (I also resolved to switch to MPW to aid this resolution) I'd take a walk around the industrial park where my day job is located. After a week of this behavior, during which I had been sacrificing my juggling practice in the quest for good health, I stepped onto the scales. You got it. I gained two pounds. I was not reinforced. Undaunted, I kept at it for another week. I didn't gain any weight that week, but then again I didn't lose any. I was neither amused nor reinforced here. After another week I lost a pound. Three weeks had passed and I lost one lousy pound. The delay between behavior and reinforcement was too long for me. During my next long compile I decided to practice with the local Nerf militia, effectively terminating my desired behavior. I wasn't reinforced fast enough, while the competing behavior of goofing off delivered an immediate reward. So it is not a lack of willpower or self-discipline but a preordained psychological behavior that keeps me in size XL World Wide Developers Conference T-shirts.

Luckily in good video games there are all kinds of reinforcement; an on-screen score, new sounds, fresh challenges, interesting levels. And all of these reinforcements are delivered instantly after the behavior is performed. Grab enough points and instantly you are transported to the next level.

Psychological Undo

All the psychological ideas that have been discussed so far are based around the principle that rewards are given for accomplishing the challenges that the game provides. Another psychological behavior that leads to further game play is the concept of regret. This is not the kind of regret that occurs after you've eaten too many pepperoni-and-pineapple pizzas. The form of regret I'm referring to is best explained by another lottery example. Yes, another one. Remember, if you don't play you can't win. Then again, the odds of winning without buying a ticket are about the same as with one.

You and a friend both buy a lottery ticket for the upcoming $300 million Programmers Sweepstakes. You both pick six numbers. Your numbers are 3, 7, 11, 23, 29, 31 (yes, they're prime, this is a programmers' sweepstakes). You friend's numbers are 3, 6, 12, 20, 31, 36. On the evening of the drawing you and your friend stop playing Donkey Kong Country (and just as you were about to finish Kong's Mines) for five minutes to tune into the drawing. Bill Gates has been asked to draw the numbered Ping-Pong balls as he's one of the few programmers who won't be upset by being disqualified from the lottery. Bill starts pressing the big red button that releases each ball into the chute. The first number is 3. The next number is 7. You're starting to get interested. Bill pushes the button and 11 falls out. You're starting to think that this could actually happen. One more button push, one more number; it's 23. Not only are your numbers coming up but they're arriving in ascending order, proving of course that God wants you to have this money. Next number . . . 29. Even your friend has stopped whining about Donkey Kong and her wasted dollar, while you're wondering if the Ferrari dealership is still open. The suspense is as thick as . . . something really thick. You

hear the slight click as Bill's index finger depresses the button. There is a momentary pause before one of the balls heads down the chute. While it's rolling you can't make out the number. Slowly it comes into focus: 8. The only thing you can think is that 8's not prime, don't they know that 8's not prime? That Bill Gates and his damn finger. No wonder he can't create a decent O. S., he doesn't even know 8 isn't prime. Poof! Gone. No pool, no Ferrari, no joining the Mac-of-the-Month club. You're crushed.

Now that's regret. The kind of regret that keeps you whining for days. But why are you so upset in comparison to your friend? You both lost only a dollar, and she's not going around telling everyone how she lost three hundred million bucks. What's your problem? Your problem is that the difference between the desired reality and the actual reality for you was only one number, but for your friend actual reality never even came close to her desired reality. The closer a desired reality is to actual reality, the greater the regret. Why? Because you know that only one thing had to change to enable the realities to match. Your friend's reality differs so greatly from actual reality that her mind is unable to construct a reasonable fantasy of how things should have been.

Again, what does this have to do with video games? A whole bunch. Video games give you the ability to instantly fix any situation that causes regret. I forget to put down the landing gear on my F-18 Hornet and end up leaving a large silicon stain all over the deck of my aircraft carrier. I have regret and lots of it. And instantly, or at least as long as it takes to get through the pilot briefing, I'm able to undo the fatal mistake. This makes me happy and I keep playing the game.

When using regret, keep in mind that you need to allow players to undo their mistakes as close to when it happened as possible. Nothing is more infuriating than getting to a point in a game that you can't get past due to a mistake made hours or even days before.

Addiction Is the Name of the Game

If you haven't noticed by now, the goal of a video game is to get the player addicted. Addicted to playing. All of the psychological prin-

ciples discussed contribute to providing a pleasing gaming experience. So pleasing that the game will become hopelessly addictive. You want your player to give up food, sleep, even personal hygiene just to spend five more minutes playing your game.

Now that you know your final devious goal with all this psychological mumbo-jumbo, let's move on.

2

Introduction to Game Animation

Arcade Games and Animation

Whenever you play an arcade game on the Mac or in an arcade, one thing is guaranteed to happen: things will be animated. Every element on the screen will be moving, bouncing, or exploding. An arcade game without animation is like . . . it's like nothing. An arcade game without animation is mah jong. Imagine Space Invaders with no invading, Pong with no ponging, Pac-Man with no pacing!? An arcade game without animation is boring. Arcade games should never (say it loud), never be boring. Repeat that with me: "Arcade games should never be boring." The best way to fight off boredom

is to keep things on the screen moving. Moving fast. This is the part of the book where you begin thinking, duh—of course games should have animation. But how am I, a poor, lowly Mac programmer, going to learn to make things animate? A good question. A good answer is in this chapter.

How Animation Works

When a Ren & Stimpy cartoon flips by at thirty (twenty-five for you readers in Europe) frames a second, can you see each frame flip as a discrete movement? What magic happens that causes Ren to appear to sneeze so smoothly? The magic is persistence of vision. Persistence of vision is what allows television, movies, video games, and Ren and Stimpy to work. You can think of this persistence as a defect in the human brain. We should only see television as thirty little pictures per second, but we don't. Each image your brain receives lingers until the next image is received. Conveniently, our brain "forgets" the gap in time between the first and second image. This forgetfulness of the brain to time-stamp each image as it comes in results in the seemingly continuous stream of images that we perceive as video, films, and animation.

Just how rapidly must a series of images change in order to fool the brain? The quality varies with the rate. If the images are presented at commercial video rates (thirty frames per second), you see an animation of good quality. Good enough for Nickelodeon must be good enough for me, right? Wrong. Get your game animation to run at thirty frames per second if you can. But if you can't, do you take your compilers and go home? No, the frame rate can drop and the brain will still pick up the pieces. How low can the frame rate go? Low. There are remnants of persistence of vision at even the pokey rate of eight frames per second but, that rate is not the goal of a game writer. If eight frames is the basement of acceptable frame rates, where's the penthouse? At this point in hardware development, most game makers view the video rate of thirty frames per second as the ultimate. Is there any point in going after a higher frame rate than 30 fps? Realism is heightened by frame

rates higher than those of television. This is evident at the StarTours ride at Disneyland. StarTours runs at the rate of 60 fps. This increased frame rate is one reason the ride is so realistic; the room full of hydraulics is the other. Still, I think it's mainly the frame rate increase. Is there a top limit for frame rates where we can't tell live from Memorex? I don't know, but I like to think the future of frame rates is similar to that of audio CD sampling rates. A CD's sampling rate of 44.1 kHz is fine for my ears and I can't tell the CD from the real thing, but some mutant's ears can. Some future frame rate will be the equivalent of the CD sampling rate, good enough to fool most of us—but not all of us. There will always be that one mutant who needs a higher rate to fool his senses. Usually these people are game critics.

I'm sure the big question on your mind is why does this persistence thing exist? If everything seen before the birth of flickering images didn't depend on persistence of vision to stitch images together, why is our brain blessed with this small anomaly? Why, in the battle of natural selection a million years ago, was this one defect kept that would allow humans to view animations in the proper way?

Video games. Isn't evolution wonderful?

Computer Animation

After computers were given the ability to display images (beyond the Snoopy calendar printouts from a DecWriter III), programmers began creating animations. Of course these animations of spaceships trying to blow each other up were built for games, but they were animations nevertheless.

Computer Graphics Flavors

Graphics in video games come in two flavors: `vector graphics` and `raster graphics`. These two techniques have formed the history of computer graphics. Vector games came first. They estab-

lished the field but began fading out in the mid '80s. Raster games arrived just before the death of vector games and are the only type of games you can put a quarter in today.

Vector Graphic Animation

Vector-graphic hardware is like a silicon Etch-a-Sketch: the only thing it can draw is lines. The classic arcade games—Asteroids, Battlezone, Omega Race, Tail Gunner, Space Duel, and my favorite, Tempest—were all created using vector graphics.

A vector graphics system's hardware draws lines that are contained in what is called the display list. Each line has its starting and ending point as an entry in a display list. The graphics hardware of a vector system grabs the endpoints of the first line to draw and then etches that line on the vector display. This line is etched when the electron beam of the display excites the points of phosphor coating on the inside of the monitor. The excited dots of phosphor cause the line to glow on the monitor. The hardware then moves on to draw the next line in the display list. When the hardware reaches the end of the display list, it returns to first line entry in the list.

Lines on the vector monitor do not last forever. Eventually the phosphor loses it excited state (a snappy line here would be too easy) and the line fades away. How fast the line fades away is determined by the type of monitor the vector system uses. To keep the image on the screen from fading away, the vector hardware must keep redrawing the same lines over and over again. If a line's endpoints are changed, the line will be drawn in its new position during the next run through the display list. It's the process of a line fading away and the hardware drawing a line in a new position that produces vector animation.

> In the category of best vector arcade game of all time, the winner is . . . darn, these envelopes always stick. Yes, it's Tempest! This game is directly responsible for my retaking linear algebra (my high score is around 610,000). Brought to you in 1981 by Dave Theurer. Produced and directed by Atari.

Interesting note: Dave Theurer went on to become a Macintosh programmer and helped create the premier graphics conversion program, Debabelizer. Debabelizer excels in converting raster files from one format to another. It's ironic that the creator of the best vector game of all time went on to a career in raster graphics. O.K., so maybe it isn't that ironic, but it's a nice tidbit for the next time you play Trivial Pursuit (the Mac programmers edition).

If you wish to follow in the footsteps of Tempest's creator, you might want to check out Juri Munkkis's game Arashi. Arashi is a fairly complete tribute to Tempest that runs on the Mac. As a bonus it comes with the complete C source code to the game and a vector animation toolkit that implements the game. You can find the Arashi package on most major on-line services.

Raster Animation

Raster animation is the flip side of the computer graphics coin. To get an idea of how raster graphics works, put your face right up to your television until your nose touches the screen. You will see nothing but lines of colored fuzzy dots. Your television screen is made up of roughly 482 lines of colored fuzzy dots. Each line has about 640 of these dots. Now take your nose away from the screen and don't forget to wipe the dust off the end.

Memory-mapped Video

Your television signal is converted to the lines of dots you see as a picture. A computer screen is made up of the same lines of dots, but in a computer they don't originate as an analog source, as your television signal does. The picture on a computer screen originates as a memory image. What you see on your computer screen is just a bunch of bit patterns stored in an area of memory referred to as raster, or display memory. Since raster displays reflect the setting of the display memory, any changes to the display memory result in almost instantaneous changes on the screen. Unlike a picture produced by vector graphics, one produced by a raster display can be much more than lines. A raster image can be anything that can be composed of tiny fuzzy dots. This flexibility comes at the cost of

memory. A vector system requires very little memory, while a raster system requires large chunks of RAM. This memory appetite is why raster-based arcade games took so long to dominate the vector game system. Arcade designers couldn't afford to build raster systems until memory became as cheap as old leisure suits. Once this price point was reached, vector games' lifetime clock ran out overnight.

The animation process for a raster system is like traditional cell animation. Instead of quickly replacing one piece of celluloid with another over a static background, a raster animation quickly replaces one chunk of memory with another chunk, combining both chunks with the existing background image that is already in the raster memory.

All Macintosh models (so far) are built on raster graphics systems. The memory for the raster display can be part of the main memory (the original Mac), on the video card (Mac II), or as separate video RAM SIMMs on the motherboards (most Macs sold today).

Three Ways to Animate Graphics on a Mac

There are three basic methods to animate graphics on a raster-based system such as the Macintosh. The techniques presented here range from the very easy to implement and use exclusive-or animation to the more difficult, yet more flexible buffered animation.

Xor Animation

The exclusive-or operation gets its name from the exclusive-or logic operator. Looking at the Table 2-1, you can see how the logic of exclusive-or works.

Table 2-1. Animation Truth Table

1 xor 0 = 1
0 xor 1 = 1
0 xor 0 = 0
1 xor 1 = 0

The above table implies the basic idea of exclusive-or animation. After drawing an image you can erase that image by drawing the image again. Not only does the image erase by your drawing it twice but the screen is restored to its previous state as if no drawing had taken place. This animation technique is used by almost all Mac applications. For an example, go to the Finder and drag out a marquee in a window. The code that animates the marquee resizing to match the mouse is being done using the exclusive-or animation technique.

To use exclusive-or animation on the Mac, you set the transfer mode of the current port to `srcXor`. Before starting the animation you must draw the object once to prime the exclusive-or cycle. Once the animation has been primed, draw the object again at its original position. This second operation will erase the object and restore the screen. Calculate the next position of the object, draw the object at the calculated position, and repeat. You have animation.

Let's look at the code that demonstrates exclusive-or animation.

```
void NextFrameXOrAnimation(Rect *bounds)
{
    PenMode(srcXor); // set the PenMode to exclusive or

    // Erase the rect at the previous frames position
    PaintRect(&demoRect);

    // Offset the rect to the next position
    OffsetRect(&demoRect, delta_X, delta_Y);

    // Check to see if rect has hit the edge of the bounds
    CheckBoundsEdges(&demoRect, bounds);

    // draw the current frame
    PaintRect(&demoRect);

    PenMode(srcCopy);  // turn off xor
}
```

First, we set the transfer mode of the current port to `srcXor`. The rectangle that is bounced around the screen is erased by painting its interior with black. The rectangle is then moved to its new position for the next frame. After the movement the rectangle is checked against the bounds of the window to see if the rectangle has hit a

window edge. If the rectangle has gone past a window edge, the rectangle is set to simulate that the rectangle has hit the edge. The rectangle is painted with black again, this time causing the rectangle to actually appear on the screen. Before leaving the function, we tidy up by setting the transfer mode to the port's default, `srcCopy`.

When you run the exclusive-or example you will see the worst thing in game animation: flickering. When animation flickers, the illusion of smooth motion is destroyed. The flickering results from the delay between erasing the previous frame and drawing the current frame. To remove the flicker in exclusive-or animation, you can either try to reduce the delay between erasing and drawing, or remove the delay altogether. Reducing the time between erasing and drawing is usually not possible without a lot of work, and work is what exclusive-or animation is supposed to save us. To remove the delay completely, you need to alter the above code sample slightly so that the erasing and drawing of the rectangle are done in one operation.

```
void NextFrameBetterXOrAnimation(Rect *bounds)
{
    Rect    prevRect, newRect;

    prevRect = demoRect;
    PenMode(srcXor);    // set the PenMode to the xor

    // Offset the rect to the next position
    OffsetRect(&demoRect, delta_X, delta_Y);

    // Check to see if rect has hit the edge of the bounds
    CheckBoundsEdges(&demoRect, bounds);

    // convert the prev position and the current position
    // to regions
    RectRgn(prevFrameRgn, &prevRect);
    RectRgn(thisFrameRgn, &demoRect);

    // get the union minus the intersection as a region
    XorRgn(prevFrameRgn, thisFrameRgn, sectFramesRgn);

    // draw the sectFramesRgn in the new position
    PaintRgn(sectFramesRgn);

    PenMode(srcCopy);   // turn off xor
}
```

To erase and draw the rectangle in a single action, we resort to using QuickDraw regions. Before we set the transfer mode to `srcXor`, we make a copy of where the rectangle is currently. As in the previous code sample, the rectangle is moved and checked against the window edges. Then previous and current frame rectangles are converted to regions by calling `RectRgn()`. The regions we pass in were built with `NewRgn()` when the demo was initialized. Another region that is the union of the two regions minus the area of overlap is created with `XorRgn()`. With one call to `PaintRgn` we get an erase of the previous frame and a draw of the current frame. By running the sample code and selecting `Better Xor`, you can see that this greatly reduces flicker compared to ordinary exclusive-or animation.

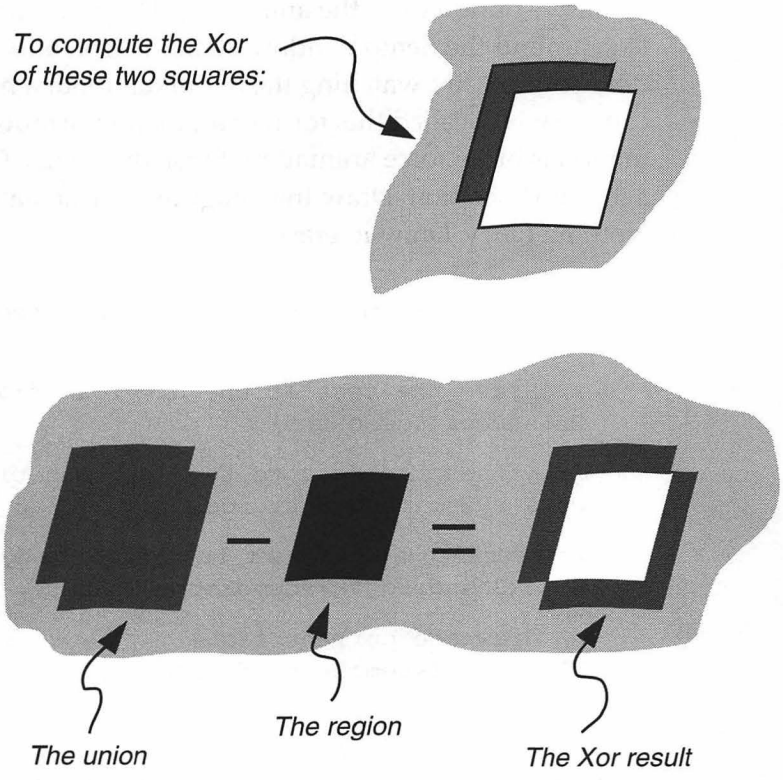


Figure 2-1. The `XorRgn` of frame one and frame two

Why would you want to use exclusive-or animation? Exclusive-or animation is memory-frugal. Other than the memory used by the object you are drawing and a region or two, exclusive-or animation is cheap. Cheap cheap. Besides being inexpensive, exclusive-or animation is pretty simple to implement. Easy to program, no memory usage to speak of. What's the catch? Catch: It's almost impossible to program more than one-element animating, and if you do, the flickering will drive you into epileptic fits. Color? Forget it, exclusive-or animation on top of 256 color backgrounds can produce some really freaky graphic side effects. With these disadvantages, about the only game that can be programmed with exclusive-or animation is Pong.

Primitive Animation

Run the sample application for this section and then choose the primitive option from the animation. The demo will animate an icon around the demo window, with the icon bouncing off the window edges. After watching the demo for about a nanosecond you can see why I called this form of animation primitive. Primitive animation is brute force animation. Erase the image. Calculate the image's next position. Draw the image at the calculated position. Nothing fancy. Draw it, erase it.

```
void NextFramePrimitiveAnimation(Rect *bounds)
{

    // Erase the rect at the previous frames position
    EraseRect(&demoRect);

    // Offset the rect to the next position
    OffsetRect(&demoRect, delta_X, delta_Y);

    // Check to see if rect has hit the edge of the bounds
    CheckBoundsEdges(&demoRect, bounds);

    // draw the current frame
    PlotCIcon(&demoRect, demoIcon);

}
```

In the demo code a rectangle is used as the frame for placing the icon on the screen. The previous frame of animation is erased. The

icon's future position is then determined. `PlotCIcon` displays the icon at its next location. Repeatedly calling this function results in some really bad animation.

To reduce the amount of flickering in primitive animation, you need to reduce the time between erasing and drawing the image at its next location. In the sample code not much time is spent between erasing and drawing. This in-between code could be completely rewritten in assembler, but you have better things to do. Optimization isn't needed here. What's needed is a better way.

Buffered Animation

Buffered animation is the promised "better way" to solve the problems of primitive animation. The erasing and drawing of the image is performed in a chunk of memory and not on screen. After the whole frame is built offscreen, or buffered, the frame as a whole is then moved onto the screen. Since the on-screen frame is replaced in one operation, the eyes see a smooth transition from one frame to the next. An erase followed by a draw is never viewed on-screen. As long as you can replace the frames fast enough, the animation will appear flicker-free.

In discussing the code for the animation style sampler, I'll be skipping the gory details. The point of this sample is to introduce you to buffered animation, not explore it in excruciating detail; we'll do that in a later chapter.

Before the buffered animation demo is run, we create the offscreen chunk of memory that we will buffer our drawing into. `NewGWorld` is a Toolbox trap provided by QuickDraw that creates what you can think of as a virtual screen. If an error is returned the demo displays the error and then bails. You'll provide better error handling than this, right? If there was no error, the parameter `offscreenGWorld` holds the offscreen buffer.

```
err = NewGWorld(&offscreenGWorld,0,
        &((**((WindowPeek)animWPtr)->contRgn).rgnBBox),
                                    NULL, NULL, 0);
if(err != noErr)
{
```

```
    PostErrorAlert(err);
    ExitToShell();
}
```

After creating the virtual screen buffer, we need to clear its contents to match that of the on-screen window. It takes these four cryptic lines just to erase the buffer. `LockPixels` makes sure that the buffer doesn't wander off while we draw in it. Through `SetGWorld`, QuickDraw is convinced to draw in to the buffer instead of the screen. Finally, the buffer is cleared by erasing it. The buffer is set free to roam the heap. To get QuickDraw to draw on-screen, once again `SetGWorld` is called. From looking at the code you can see that with buffering you spend most of the time getting QuickDraw to draw into the buffer instead of on the screen. Well, you do. Get used to it.

```
// erase the offscreen background
LockPixels(offscreenGWorld->portPixMap);
SetGWorld(offscreenGWorld, NULL);
EraseRect(&thePort->portRect);
UnlockPixels(offscreenGWorld->portPixMap);

// reset the current graphics environment
SetGWorld(currPort, currDev);
```

Enough setup, let's bore into the animation code. Before we start animating in the offscreen buffer, the QuickDraw's current drawing environment is retrieved with a call to `GetGWorld`. After locking down the pixel, QuickDraw is then told to draw into the buffer that was created earlier.

```
void  NextFrameBufferedAnimation(Rect *bounds)
{
    GDHandle      oldGDh;
    CGrafPtr      oldPort;
    Rect          prevPosition;
    Rect          oldAndNewPosition;

    // remember the current graphic world
    GetGWorld(&oldPort, &oldGDh);
```

```
// prepare the offscreen for drawing.
// Don't want our pixels wanderin' off
LockPixels(offscreenGWorld->portPixMap);

// Make the offscreen the current port for
// QuickDraw to draw quickly in
SetGWorld(offscreenGWorld, NULL);
```

Enough setting up QuickDraw, let's animate. As in the primitive animation code, the icon's former position is erased and the future location of the icon is calculated. The icon is then drawn at this position. Nothing different here from the primitive example other than that nothing is shown on-screen. QuickDraw is redirected to the drawing environment that was remembered earlier. The whole buffer is then blasted on-screen in one shot with CopyBits, the workhorse of buffered animation. After moving the frame to on-screen, we let the buffer's pixels roam with `UnlockPixels`.

```
// Erase the rect at the previous frames position
EraseRect(&demoRect);

// Offset the rect to the next position
OffsetRect(&demoRect, delta_X, delta_Y);

// Check to see if rect has hit the edge of the bounds
CheckBoundsEdges(&demoRect, bounds);

// draw the current frame
PlotCIcon(&demoRect, demoIcon);

// Done drawing in the offscreen so reset back to the
// previous port and graphics world
SetGWorld(oldPort, oldGDh);

// Copy from the buffer to the screen
CopyBits((BitMap *)(*(offscreenGWorld->portPixMap)),
                  &thePort->portBits,
                  &thePort->portRect,
                  &thePort->portRect,
                  srcCopy, NULL);

// let the offscreens' bits roam
UnlockPixels(offscreenGWorld->portPixMap);
```

The buffered animation looks much better than the primitive animation, but it also runs slower. Much slower. This slowdown is due to the huge amount of pixels CopyBits is moving from the offscreen buffer to the screen. At 256 colors with a window that is 640 pixels by 480 pixels, CopyBits is having to haul 307,200 bytes around per frame. It takes some time to drag that many pixels around. The best way to increase the speed of buffered animation is reduce how much work CopyBits has to perform.

The better buffered example code shows how to reduce the amount of pixels CopyBits has to move by moving only the bits that have changed from one frame to the next. To determine the minimum amount of pixels that need to be moved, we make a copy of the rectangle that defines where to put the icon. The icon rectangle is then offset as in all the previous animation samples. Before we call CopyBits we get a rectangle that is the union of the area we erased and the area that was just drawn. This union area is much smaller than the whole screen, which results in a lot fewer pixels being moved. With the rectangle at 35 pixels by 35 pixels and the screen at 256 colors, we are moving 1225 bytes. Wow, 1225 bytes versus 307,200 bytes, and all this by adding only two lines of code. I love programming.

```
// Remember where the icon was
prevRect = demoRect;

// Offset the rect to the next position
OffsetRect(&demoRect, delta_X, delta_Y);

// Check to see if rect has hit the edge of the bounds
CheckBoundsEdges(&demoRect, bounds);

// draw the current frame
PlotCIcon(&demoRect,demoIcon);

// Done drawing in the offscreen so reset back to the
// previous port and graphics world
SetGWorld(oldPort,oldGDh);

UnionRect(&demoRect, &prevRect, &blitRect);

// Copy from the buffer to the screen
CopyBits((BitMap *) (*(offscreenGWorld->portPixMap)),
            &thePort->portBits,
            &blitRect, &blitRect,srcCopy, NULL);
```

Memory Costs

If buffered animation produces the smoothest display, why would you even bother with any other method? 'Cause there's no such thing as a free lunch. Buffered animation has a very high cost when it comes to memory. Compared to exclusive-or animation, the buffered technique is an absolute memory pig. Since you'll be using buffered animation for the majority of the animation in this book, let's look at what the memory cost will run us.

To determine how much memory a pixel buffer will use, you need to know the height, width, and number of colors you plan to store in the buffer. The typical buffer you will use will match the screen's attributes, so we'll use that as the example here. The average color monitor is a 13-inch screen, which has a height of 480 pixels and a width of 640 pixels. The number of colors or shades of gray that can be displayed usually runs from straight black and white, increases to 16 colors, and usually stops at 256 colors or grays. The formula for how much memory a buffer will use is simple: multiply the height times the width times the pixel depth expressed as bits or bytes. Table 2-2 gives the memory usage for a 13-inch screen at all possible color depths.

Table 2-2. Memory Usage for a 13-Inch Screen

Width	*Height*	*Pixel Depth*	*Memory Usage*
640	480	1 bit (black & white)	38,400 bytes
640	480	4 bit (16 colors)	153,600 bytes
640	480	8 bit (256 colors)	307,200 bytes
640	480	16 bits (Thousands of colors)	614,400 bytes
640	480	32 bit (Millions of colors)	1,228,800 bytes

At 256 colors, the depth we'll be using for most of this book, the buffer will eat up close to 300K of memory. This is where most programmers over thirty immediately launch into war stories of how they had to program the entire Apollo space program in an 8K ROM and could only use chopsticks to toggle the bits in. These

stories always start when you talk about using any more memory than 16K. Ignore them, or if you are over thirty quit telling them. Memory is cheap and plentiful. Games on the Mac take that to heart in their memory appetites. A Mac game that runs in 256 colors will usually run in a memory partition of 1 MB of memory or even more. Buffered animations do use a lot of memory, but buffering the drawing is the easiest way to create professional-looking games.

Performance Costs

You really can't do much about how much memory your games will use, so let's move on and look at something you can do something about—the speed of moving all those bits.

Buffered animation has two strikes against it, memory and speed. We'll ignore memory usage for now, but the speed of the animation is something you have control over. Speed is measured in how fast the computer can move the bits from your offscreen buffer to on-screen. The slower that the bits are moved, the slower the frame rate of your animation. It's easy to underestimate how long it takes to move a buffer full of pixels, since it takes only one call to CopyBits and your offscreen bits are on-screen bits. If it takes only one line of code to have those bits moved, it's easy to forget how long CopyBits can take to move them. Glaciers have melted in less time than it takes CopyBits to transfer a screen full of pixels. Reducing this glacial aspect of buffered animation is the major goal of any game programmer.

Reducing how long it takes to copy or blit pixels from offscreen to on-screen is the holy grail of Mac game programming. About the only thing limiting the speed of buffered animation is how fast the buffer be can moved. If you want to increase your animation's frame rate, reduce your time in CopyBits. Speeding up blitting can be done in two ways on the Mac. Use CopyBits ideally or write your own replacement for CopyBits that is optimized strictly for game programming.

If you've ever looked at a spec sheet for a Super Nintendo or a Sega Genesis, you've seen that these systems have at their cores some puny little microprocessors compared to our wonderful Macintoshes. So how can these tiny processors produce such fast-paced games with seemingly hundreds of things going at once? Hardware. Game systems and even some computers have dedicated hardware for shuffling pixels around. This dedicated hardware is usually referred to as a blitter. The little microprocessor only has to tell the blitter where the pixels are and where it would like them, and bang, the pixels are moved. That's all a blitter can do, blit pixels. No adding, no subtracting, just blitting, very fast blitting.

Why Buffered Animation?

For the rest of the examples in this book we will have to use one of the forms of animation presented. The Mac does not have vector hardware support, so that decision is an easy one. Exclusive-or animation cannot support the complex animations that appear in arcade games, so strike exclusive-or as a possible candidate. Primitive animation produced so much flashing that you'd have to tie your players down to a chair to force them to play your game. That leaves buffered animation as the only candidate for the rest of the book. In spite of the memory and performance cost, buffered animation is the choice of all Mac arcade games. The rest of the book will be dedicated to building and using buffered animation as the main means of moving your game elements.

3

Offscreens and the Mac

Offscreens allow us to redirect where QuickDraw renders its images. QuickDraw operations are normally visible on the screen as they execute. `Call Line()` with the parameters 10, 10 and you will see on-screen a diagonal line from the current pen position that is ten pixels long. All QuickDraw does is set the appropriate pixels in the chunk of memory that is the video memory. This video memory can exist on a NuBus card, a section of video RAM (VRAM) on the motherboard, or just an area of memory reserved for the screen display. The Mac hardware takes care of turning the video memory into signals that your monitor can display. QuickDraw doesn't know or care which chunk of memory it's drawing in. As long as

the memory is described in enough detail, QuickDraw can image into the buffer. In this section we will build a library that will allow us to allocate, dispose, and manage offscreen buffers on the Macintosh. This library will become the foundation of all our future game programming.

How QuickDraw Sees Memory

QuickDraw is the raster graphics package that operates on Macintosh displays. As we saw in Chapter 2, a raster display is one that maps the pixels contained in an area of memory into analog signals needed to display the images on the monitor. We're interested in the area of memory that QuickDraw draws or rasterizes into. This area of memory is called a frame buffer, or video memory. QuickDraw slices this memory into a coordinate system with the origin located in the upper left of the buffer, with x coordinates increasing to the right and y coordinates increasing downward (see Figure 3-1).

The height and width of the video memory establishes the boundaries QuickDraw will draw within. If you ask QuickDraw to color a pixel whose coordinates lie outside the buffer's extent, it will just ignore the request. If QuickDraw didn't ignore these requests, it would be changing memory that doesn't belong to the frame store. If this memory doesn't belong to the frame store, it belongs to something else, probably your program. Indiscriminately writing into memory that doesn't belong to you is a quick way to crash a Mac. When QuickDraw knows the dimensions of the frame buffer, it is able to overlay a coordinate system on top of the memory in the video buffer. With this coordinate system in place, a program can then address each pixel in the video buffer.

For QuickDraw to turn a pixel black at location 0,0 (the upper left hand corner of the screen), QuickDraw sets the bit that maps into the row 0, column 0 to 1. This assumes that one pixel equals one bit in the video buffer, just like the original black-and-white Macs. The video circuitry will eventually (a microsecond later) display the black pixel on the screen. For QuickDraw to color a pixel, it needs to convert the x, y coordinate of the pixel to the location of

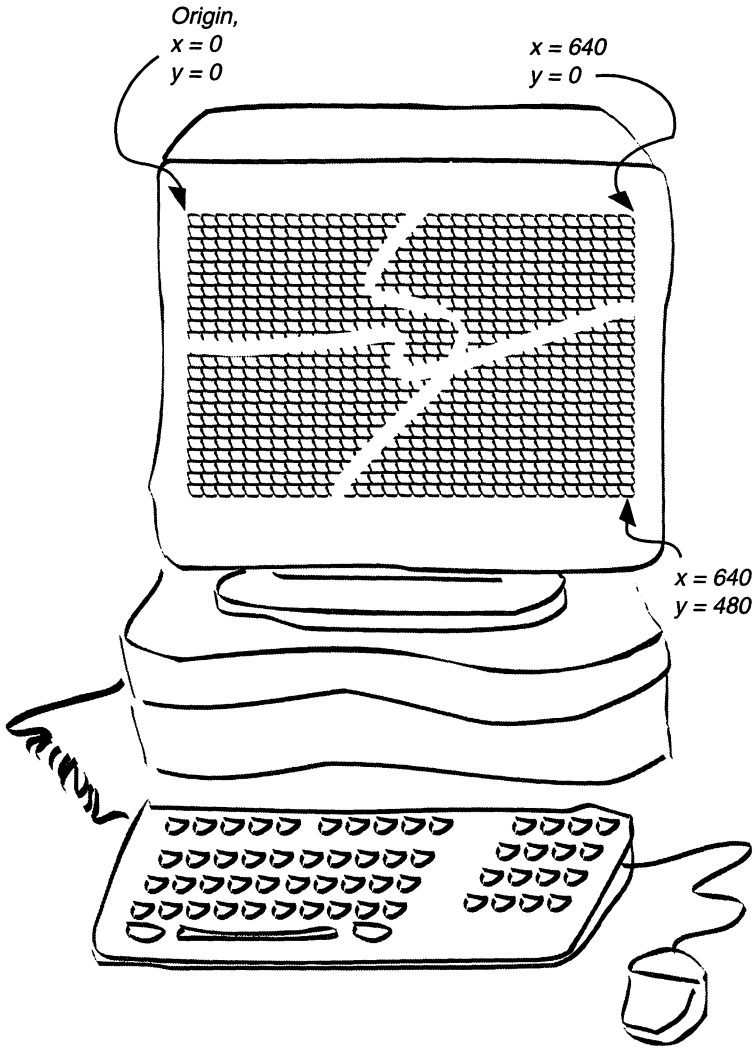


Figure 3-1. The pixel layout in a frame store

the bit in the video buffer's memory. To accomplish this mapping from graphical coordinates to memory locations, QuickDraw needs to know the dimensions of the frame buffer and how many bytes it has to skip in the video memory to get to the next row. Supplied with this information, QuickDraw can cleanly draw without going outside the lines.

Offscreen Elements

In order to get QuickDraw to use our offscreen buffers, we need to convince it that our offscreen buffer is an appropriate place for it to draw in. Getting QuickDraw to image in our buffers depends on what version of QuickDraw we use. These samples will work with black-and-white QuickDraw, referred to as Classic QuickDraw, and Color QuickDraw. Color QuickDraw is the version of QuickDraw that appeared with the introduction of the Mac II. The last version you'll work with is 32-Bit QuickDraw, which was introduced to allow the Mac to deal with more colors than 256. This version of QuickDraw was once referred to as TrueColor QuickDraw. Each version of QuickDraw requires a slightly different approach to get it to use the buffers. The goal of our offscreen library is to hide the differences needed for each version of QuickDraw and present one interface that will work with all versions of QuickDraw.

Classic Offscreens

In the QuickDraw version that originally shipped with the first Macs, QuickDraw could only draw in black and white. To use offscreens with the original version of QuickDraw, you need to understand two parts, bitmaps and grafPorts.

Bitmaps

In order to use black-and-white offscreen, you need to understand how to talk the talk of Mac offscreens. And the first word of the language is bitmaps. The techno definition is below.

```
struct Bitmap{
    QDPtr    baseAddr;
    short    rowBytes;
    Rect     bounds;
};
```

baseAddr Ptr

The bitmap structure provides a coordinate view of a chunk of memory. The memory chunk is usually referred to as the bit image, or the pixel image. The baseAddr field points to the chunk of memory that the bitmap will use as its bit image. This bit image is where QuickDraw will work its graphical magic. This field can point to anywhere in the Mac's memory map. For offscreens, its baseAddr will usually point to a piece of memory in main memory allocated by a call to `NewPtr`.

> You can have any number of bitmaps pointing to the same area of memory through their baseAddr fields. This is the foundation of how the Mac displays multiple windows on one screen. Each window has in it, deeply buried, a bitmap with the bitmap's baseAddr pointing at the video buffer of the screen. There's plenty of other programming voodoo that makes windows on the Mac work, but the basis is laid with bitmaps all pointing to the same area of memory.

Bounds Rectangle

The bounds rectangle field of a bitmap overlays a coordinate system on top of the bit image pointed to by baseAddr. The bounds rectangle doesn't have to include all the pixels contained in the bit image. Besides providing a coordinate system, the bounds field establishes the dimensions of the bitmap. With the bounds rectangle

QuickDraw can calculate the height and width of the bitmap. When QuickDraw knows where the bit image is and how to address each pixel, it has almost enough information to draw in the bitmap. QuickDraw needs only to know the `rowBytes` to have full access to the bitmap.

rowBytes

The rowBytes field is a small bit of information that QuickDraw needs to draw into a bitmap. With the rowBytes filled, QuickDraw can determine how many bytes a bit image has in each row. rowBytes is an optimization field that QuickDraw uses to get from one row to the next even though the bit image is in a linear chunk of memory. Without the rowBytes field, QuickDraw would have used the width of the bounds rectangle to determine how many pixels to skip to get to each row. This would be slow. Worse than that is the case where the width of the bitmap doesn't fit evenly into a number of bytes. Here there would be pixels that are at the extreme right and the extreme left of the bit image and yet would be contained within one byte. This would be terribly slow. To make sure this split byte never happens, QuickDraw uses the rowBytes field instead of depending on the bounds rectangle. Using rowBytes does mean there will be a small amount of memory at the right edge of the bit image that is unused. A little memory loss for a big gain in speed is what QuickDraw believes in. QuickDraw also requires that rowBytes be an even number of bytes.

Building a Bitmap

Now that you know all the dirty details of QuickDraw bitmaps, let's look at the programming steps it takes to build a bitmap.

The first step is to create the bit image that will hold the pixels. The bit image that the baseAddr points to must be large enough to hold all the pixels. QuickDraw needs rowBytes to be big enough to contain a row of pixels and be an even number of bytes. This requirement prompts the weird math that divides the width of a row of pixels by 16, adds one to make sure the rowBytes will be even,

and then multiplies the total by two to get the number of bytes. Let's make sure this works with our sample numbers. We'll use as an example a bitmap width of 53 pixels.

53 / 16 = 3, 3 + 1 = 4, 4 ˇ 2 = 8

Figure 3-2. Weird math

Eight as a value for rowBytes would give you room for 64 pixels, which is the closest number that satisfies QuickDraw needs of having rowBytes be big enough to hold all the pixels and yet be even. Yea! The math works.

```
BitMap MakeBitMap(short wantedHeight,
short wantedWidth)
{
   BitMap aBitMap;// the bitmap we'll create and return
   short     numberRows; // number of rows in the bitmap
   short     tempRowBytes;
   Rect      bitMapBounds;

   tempRowBytes = ((wantedWidth / 16) + 1) * 2);

   // create the bit image, we cast up to longs so we
   // don't get an overflow at 32,767 pixels
   aBitMap.baseAddr = NewPtr((long)tempRowBytes *
                        (long)wantedHeight);

   // you should check for failure to get a pointer and do
   // some appropriate error handling

   // set up the rowBytes and the bounds fields of the
   // bitMap
   aBitMap.rowBytes = tempRowBytes;
   SetRect(&aBitMap.bounds ,0, 0, wantedWidth,
                                  wantedHeight);

   return (aBitMap);
}
```

After the size of rowBytes is determined, a bit image is created that is the number of rows times the value stored in rowBytes. Since rowBytes and the bitmap height are both shorts, you need to cast

them both up to longs or else you could get an overflow when you request a bit image that is bigger than 32,767. This is a bug that has bitten me many times, so don't forget those casts. No error checking is done to verify that the memory request was fulfilled. You would never do this, right?

After building the bit image you need to fill in the other fields of the bitmap. You have already calculated the rowBytes, so just assign the value to the bitmaps rowBytes field. The bitmaps bounds is set to be a rectangle that upper, left is at 0,0 and has the width and height that was passed in to the function. I like my bitmaps aligned to 0,0 because it seems neater, but you could have set the bounds to any rectangle that has the correct height and width. After you assign the bounds rectangle, the bitmap is complete and you are done.

GrafPorts

In order for QuickDraw to draw in a bitmap, it needs a graphic environment wrapped around the bitmap. This graphic environment is called a grafPort. I like to think of grafPorts as the studio in which QuickDraw works, and a bitmap as QuickDraw's canvas. That's really the way I think. Here's the propeller-head description of a grafPort.

```
struct    grafPort    {
    short         device;
    BitMap        portBits;
    Rect          portRect;
    RgnHandle     visRgn;
    RgnHandle     clipRgn;
    Pattern       bkPat;
    Pattern       fillPat;
    Point         pnLoc;
    Point         pnSize;
    short         pnMode;
    Pattern       pnPat;
    short         pnVis;
    short         txFont;
    Style         txFace;
    char          filler;
```

```
    short         txMode;
    short         txSize;
    Fixed         spExtra;
    long          fgColor;
    long          bkColor;
    short         colrBit;
    short         patStretch;
    Handle        picSave;
    Handle        rgnSave;
    Handle        polySave;
    QDProcsPtr    grafProcs;
};
```

Most of these fields hold the status of the drawing environment and are not really important to you at this point. See *Inside Mac*, volume I, page 165, for the in-depth skinny on grafPorts. The field that's important to you is the portBits structure; it's the bitmap that grafPort is associated with. Every grafPort structure has a bitmap field. The port's (grafPorts are called ports for short) bitmap is where QuickDraw will do its job. When a grafPort is created and initialized with a call to OpenPort(), portBits has a copy of the bitmap of the main screen.

Now you have both parts of offscreens, the grafPorts and the bitmaps. You need to know how to get these two to work together. It seems simple enough. Create a grafPort. Create a bitmap that will hold your offscreen drawing. Then somehow get the grafPort to use your bitmap instead of the screen's bitmap. This isn't very hard because the smart people at Apple already did the hard part. Call SetPortBits() with your bitmap and the current port will then use the passed bitmap. Easy, as the code shows.

```
BitMap        offscreenBitmap;
GrafPort      offscreenPort;
GrafPtr       prevPortPtr;

// remember the original port
GetPort(&prevPortPtr);

// open our port
OpenPort(&offscreenPort);
```

```
// create our offscreen bitmap, 30 pixels wide and high
offscreenBitmap = MakeBitMap(30,30);

// Let the offscreen port use the offscreen bits
// First make the offscreen port the current port
SetPort(&offscreenPort);
SetPortBits(&offscreenBitmap);

// Now all QuickDraw calls will be drawing in our offscreen
```

Color Offscreens

Before 1987, programming a Mac was pretty simple. Not only could you count on the size of the screen, you could count on the user's having a fixed number of screens—one. Color? Who needed color, you already had two—black and white. But that was 1987. Then Apple decided that Mac programmers were getting lazy. To shape up the programming masses Apple unleashed the Mac II, and programming offscreens was never simple again. Two colors. Gone. You could now program with 256 colors. And not just 256 Apple-approved colors. You could pick any 256 colors out of 16 million choices. One screen. Gone. You could have up to six screens. The screens could be huge or small. To program in this new Technicolor world you have to understand the color equivalents for bitmaps and grafPorts: pixmaps and color grafPorts. The year 1987 also introduced graphic devices that have no black-and-white ancestor.

Here a Color, There a Color

How Color QuickDraw manipulates the raster memory to generate colors on-screen must be explained before we jump into color offscreens.

Color QuickDraw works in the RGB color space. A color space is a geometric view of the space as mapped out by colors. The RGB color, which is short for red, green, and blue, space is a three-dimensional cube with each color representing one of the three axes. Every point in this cube of color is a unique color that is a mixture of red, green, and blue. The color black is at one corner of

the cube and has the RGB value of 0,0,0. White is located at the diagonally opposite corner from black, and its RGB values are fully saturated. Color QuickDraw deals with RGB values through the `RGBColor` structure.

```
typedef struct RGBColor {
      unsigned short red;
      unsigned short green;
      unsigned short blue;
} RGBColor;
```

The `RGBColor` structure is a high-precision definition of a color, with each component ranging from 0 . . . 65,535. Color QuickDraw internally scales down the precision of each component from an unsigned integer to an unsigned byte. By scaling each component down to a byte, Color QuickDraw can access over 16 million colors. The RGBColor structure allows QuickDraw to maintain precision through various color operations without any visible changes on-screen. You will be working with RGB colors exclusively in the game programming that follows.

> The RGB color space is not the only color space in use with computers. Some of the other color spaces you could dabble in are HLS (hue, lightness, saturation), CMYK (cyan, magenta, yellow, and black), and HSV (hue, saturation, value). Each color model has its benefits and disadvantages. Some are great for dealing with color printers, while others are better at matching color photography. For games, the best color space to work in is RGB. Oh, if you can explain why CMYK is an acronym for cyan, magenta, yellow, and black, please drop me a note.

In 1987, building a Mac that could display all 16 million colors at one time seemed out of the question. Memory was just too darn expensive. A 640 ˘ 480 frame buffer capable of displaying all those colors would need 1,228,800 bytes of RAM. This would have been more RAM than was shipped with the original Mac II. And 1987 was also a time when the U.S. government helped make a

megabyte of RAM run over $400. Since memory was scarce and thus expensive, Apple decided that we would only need to see 256 colors at one time on-screen. After three years of being served only cold, stale black and white, while the dreaded PC crowd had a paltry choice of 16 colors, 256 vibrant colors seemed like a feast too good to be true. To achieve this feast without using over a megabyte of memory, Color QuickDraw uses a technique called indirect colors.

A bit of background first. Apple cut the amount of video memory that Color QuickDraw used for a screen 640 ˘ 480 from over a megabyte to 307,200 bytes. Of course they also cut back the number of colors from millions to 256. This was done by using indirect colors. Indirect colors works on the principle that instead of storing the actual RGB values in the raster memory, why not just store a smaller number—a number that uses less memory and can point to the RGB color that will appear on-screen. When Color QuickDraw is displaying 256 colors, the size of the entry is one byte. This byte is used as an index into a table of 256 RGB colors. Say the first item in the RGB array is red (65535, 0, 0) and the next is a nice vermilion (59628, 11236, 65535). To put a pixel of red on the screen you write a byte of value zero into the raster memory. For vermilion the byte would have a value of one. The values that are written into raster memory are called indirect colors. Indirect colors work like pointers in C. The size of the entry in the raster memory can range from one bit to one byte. With a one-bit entry size, QuickDraw assumes that you only want the colors black and white and hence there is no corresponding RGB table. The entry size, for reasons of speed, is forced to be in powers of two. This gives Color QuickDraw the ability to draw in black and white and in 4, 16, and 256 colors. Each smaller entry size means a smaller amount of raster memory is needed. This economy of memory and the fact that a processor can move one byte faster than three is why Color QuickDraw has indirect colors.

When an indirect color is used, the place the color indirectly points at is a color table. A color table is an array of RGB colors. Let's look at the structure definition of a color table and then go over the few esoteric details of the structure.

```
typedef struct ColorTable {
      long        ctSeed;
      short       ctFlags;
      short       ctSize;
      CSpecArray  ctTable;
} ColorTable, * CTabPtr, ** CTabHandle;
```

The ctSeed field is used as a unique identifier so that Color QuickDraw can quickly tell that one color table is different from another. Without the use of the ctSeed, Color QuickDraw would have to compare each table by comparing its RGB colors with the others'. This would be so slow that you would think you were running Windows. Whenever a color table is created by Color QuickDraw it will make sure the ctSeed is set correctly. If you handcraft a color table, you will need to call `GetCTSeed` to make sure you get an Apple-approved value for use in the ctSeed field.

The `ctFlags` field is currently used only for distinguishing who owns a color table. If the high bit of the flag field is set (the only flag defined at this point), the color table is owned by a graphics device. For other color table uses, make sure this high bit is cleared.

The `ctSize` field is a zero-based count (number of entries minus one) of the number of RGB color entries in the color table.

The `ctTable` is where the array of RGB colors is located. Of course this field couldn't just be a simple array of RGB color structures. No, that would be too simple. Color QuickDraw needs to define yet another structure. O.K., here is the definition of a CSpecArray.

```
typedef struct ColorSpec {
      short       value;
      RGBColor    rgb;
} ColorSpec, * ColorSpecPtr;

typedef ColorSpec CSpecArray[1];
```

For many years I have been able to safely ignore what the `value` field of a ColorSpec is actually used for. If you really must know, check out *Inside Mac*, volume V, page 137. But trust me, for

game programming you can remain ignorant of this field and just use the RGB field as God intended.

Now that you know a ColorSpecArray can just be thought of as an array of RGB color structures, you can move on to where the real fun is in color tables—inverse color tables.

With indirect colors, color tables have two purposes. The first use is, if given an indirect color, what is the RGB value that corresponds to that color? This seems pretty easy, as the indirect pixel value is an index into the ctTable field of the color table. The second use of color tables is the inverse of the first use. If given an RGB value, what index value in the color table comes closest to matching the RGB color? The simple way to do this is to look through all the entries of RGB colors in the table and return the index that came closest to matching the requested RGB color. This process is very slow. Speed wouldn't be a problem if Color QuickDraw only did this once a day. The trouble is Color QuickDraw can sometimes use this look-up process thousands of times in one graphic operation. To avoid being compared to a Commodore 64, Color QuickDraw uses a structure called an inverse color table to speed up this process. Here is the structure definition of an inverse color table.

```
typedef struct ITab {
      long              iTabSeed;
      short             iTabRes;
      unsigned char     iTTable;
} ITab, * ITabPtr, ** ITabHandle;
```

An inverse color table is a dictionary for color tables. Given an RGB value, Color QuickDraw uses it to look up the indexed color value in the inverse color table. To understand how this search process works let's look at how an inverse color table is built.

When building an inverse color table, Color QuickDraw takes each RGB value from the color table and processes that RGB value to produce an index into the inverse color table's `iTTable` array.

At the proper entry in the inverse color table, Color QuickDraw stuffs the indexed color value that matches the RGB value it is currently inverting. Color QuickDraw processes the RGB value by concatenating so many upper bits of each of the red, green, and blue components together to build a 16-bit word. How many of the upper bits extracted from each color component used to build that 16-bit word is set in the `iTabRes` field of the inverse table structure. Valid values for the `iTabRes` field are three, four, and five bits. Based on the bit size or resolution of this field, you can determine how large the inverse table is going to be. For a three-bit resolution the table size would be 2^9, or 1024 bytes. Four- and five-bit resolution tables would have sizes of 4K (2^{12}) and 32K (2^{15}). The default resolution for inverse color tables is four bits. You might have noticed that since the inverse table is built by lopping off the insignificant bits of the RGB color, there might be a problem with RGB values that are only differentiated by those lower bits. And you would be right. Those colors that can only be told apart from their lower bits are referred to as hidden colors. So how does QuickDraw solve the hidden color conundrum? Got me, and Apple isn't telling. *Inside Mac* just refers to some proprietary information appended to the end of the inverse color table that is used to find the hidden colors. One last detail of inverse color tables is the `iTabSeed` field. This field is just a copy of the `ctSeed` field of the color table that is used to build the inverse color table.

Pixmaps: A Technicolor Bitmap

Now that you have an understanding of what color tables are and how they work within Color QuickDraw, you are ready to explore the color side of bitmaps—pixel maps. A pixel map, or pixmap for short, is used for exactly the same purpose as a bitmap is in classic QuickDraw, to hold a pixel image. Here is the structural definition for a pixmap. You'll notice that a pixmap structure has a few more fields than a bitmap. Twelve more fields, to be exact. Makes you wonder if writing color games is worth the effort.

```
typedef struct PixMap {
      Ptr          baseAddr;
      short        rowBytes;
      Rect         bounds;
      short        pmVersion;
      short        packType;
      long         packSize;
      Fixed        hRes;
      Fixed        vRes;
      long         planeBytes;
      CTabHandle   pmTable;
      long         pmReserved;
} PixMap, *PixMapPtr, ** PixMapHandle;
```

Just Like a Bitmap, Kinda

The first three fields match the three fields that make up a bitmap. Again the `baseAddr` field is a pointer to the chunk of memory where the pixel image is stored. Like the `baseAddr` field, the `bounds` field has not changed from its bitmap counterpart. The bounds field still inscribes a coordinate system on top of the pixel image. Unlike the first two fields, the `rowBytes` field has changed slightly from the bitmap version. The `rowBytes` field still contains the number of bytes that are contained in one row of the pixmap. Where a bitmap was limited to a `rowBytes` value of $2000 or less, a pixmap can have a `rowBytes` value up to $4000. In a bitmap the upper two bits must be cleared. When building a pixmap the most significant bit of `rowBytes` must be set. Color QuickDraw uses these upper bits of `rowBytes` as flags to determine if a passed-in parameter is a pixmap or bitmap. By using these flags Color QuickDraw lets you use the same Toolbox calls you used with ordinary bitmaps. As before, `rowBytes` must be an even number, but with Color QuickDraw if you force `rowBytes` to be a multiple of four, your graphic operations will be snappier. You don't have to, but your graphics will be slower. You can't possibly want that.

How Big Is That Pixel?

With bitmaps QuickDraw knows that one pixel is equal to one bit. Pixmaps don't have this luxury. Since pixmap pixels can vary in size from one bit up to 32 bits, the pixmap structure needs some fields that inform Color QuickDraw how many bits make up a pixel. These fields are `pixelSize`, `cmpCount`, and `cmpSize`.

The `pixelSize` field contains the number of bits that make up each pixel. This can be any value as long as it's either 1,2,4,8,16, or 32. For most of this book you will be using either 4 or 8 bits as the pixel size for your pixmaps.

The `cmpCount` field, which is short for component count, contains the number of color components that make up one pixel in the pixmap. In an indexed color system the pixel is represented by an index into a color table. This would give you a component count of one. This value of one is good for pixel sizes up to 8 bits. At this time the only other value that is legal for this field is three. Three would be the setting if your pixel size was either 16 or 32 bits. Since you'll be programming games that only use a pixel size of either 4 or 8 bits, you can safely set the `cmpCount` field to one.

The `cmpSize` field, which is short for component size, holds the actual size of each pixel. Don't confuse this with the `pixelSize` field. I did. It has a different purpose for pixel sizes greater than 8 bits. And since you won't be dealing with pixel sizes greater than 8 bits, you can safely set this field to be the same value as the `pixelSize` field. You can look up the in-depth definition for `cmpSize` in *Inside Mac*, volume V.

Are Your Pixels a Little Chunky?

Up till now you have been exposed to only one type of pixel layout in pixmaps—chunky pixels. Chunky pixels are not pixels that need to cut back on the Cheetos and Jolt cola. Chunky pixels are pixels in which the bits that make up the pixel are laid out consecutively in memory.

The other pixel layout types are planar and planar/chunky. The `packType` field is where you set what style of pixel layout your

pixmap will use. In a planar layout the bits that make up a pixel are spread across the several planes of bits. The number of planes is equal to the number of bits that comprise a pixel. The `planeBytes` field is the offset in memory needed to jump from one bit plane to the next. A neat system, one that can be very useful for game programming. Trouble is, Color QuickDraw does not support planar pixel images.

Combine chunky pixels and planes of pixels and you get chunky/planar. Chunky/planar pixel images have each of the color components of an RGB pixel split across three planes, one for each color component. The field `planeBytes` is again the offset needed to get from one color plane to the next. Again, this sounds useful, again Color QuickDraw disappoints. At this point Color QuickDraw does not support this layout style (yet). *Inside Mac* can be such a tease at times.

PixMap Leftovers

Before leaving the fascinating world of pixmaps, let's quickly go over the remaining fields of a pixmap.

- `pmVersion` The version number of the pixmap, which is normally zero. If the pixmap's `baseAddr` field is 32-bit clean and cannot be used as 24-bit address, then pmVersion should be set to 4. All other bits of the version are reserved and are not to be touched.
- `hRes, vRes` A pixmap's resolution is measured in the number of pixels per inch. The two resolution fields being fixed point number allows for fractional resolutions in a pixmap. The default resolution of a pixmap is 72.0 pixels per inch (ppi) in both directions.
- `pmTable` The `pmTable` field holds a handle to the color table used by your pixmap.
- `pmReserved` To make sure that your pixmaps work in the future, make sure you set `pmReserved` to 0.

Graphic Devices

First, let me get this off my chest. Graphic Devices, or GDevices, confuse me. Actually, GDevices confuse almost everybody. The chapter in *Inside Mac* on GDevices isn't of much help; the Book of Revelations is more easily interpreted than this section of the usually lucid *Inside Mac*. I hope to fix that. At least, I hope to educate myself on just what these GDevices are and why the heck a game programmer would care.

Just What Does a GDevice Do?

When Apple created Color QuickDraw they threw in the capability to have QuickDraw use more than one display monitor. Along with the ability to use more than one monitor Apple allowed hardware manufacturers other than Apple to build these extra displays. Apple was smart enough to realize that if you allow three different hardware manufactures to build three different display cards the programmers would have to support, you guessed it, you'd get three different software interfaces. This situation already existed in the DOS market, and the endless grief DOS programmers have to go to in order to support these various displays inspired Graphic Devices. Here is the structure definition for a GDevice.

```
typedef struct GDevice {
      short             gdRefNum;
      short             gdID;
      short             gdType;
      ITabHandle        gdITable;
      short             gdResPref;
      SProcHndl         gdSearchProc;
      CProcHndl         gdCompProc;
      short             gdFlags;
      PixMapHandle      gdPMap;
      long              gdRefCon;
      Handle            gdNextGD;
      Rect              gdRect;
      long              gdMode;
```

```
        short               gdCCBytes;
        short               gdCCDepth;
        Handle              gdCCXData;
        Handle              gdCCXMask;
        long                gdReserved;
} GDevice, * GDPtr, ** GDHandle;
```

A Graphic Device is an abstract interface that describes what Color QuickDraw can image into. The device part of GDevice is a misnomer. The device can be an actual device such as a display monitor or a printer. The device can also be a nondevice such as an area of memory used for a color offscreen. By having programs and Color QuickDraw always work through GDevices, the user can switch, add, or delete monitors and the software still works.

> If you ever find yourself in a death duel of "my computer platform is better than yours," remember to take advantage of GDevices and drag a window across several monitors. If you can, make sure each monitor is a different color depth and at least one is set to black and white. When your combatant finally notices that one window is stretching across all the monitors and the Mac is doing depth conversion on the fly, you will have delivered the coup de grace. I've personally had several diehard Windows users burst into tears at the sight of this. Remember to only use this maneuver as a last resort.

A GDevice's responsibilities include describing the pixel arrangement the device will use, providing the bottlenecks for color mapping, and supplying the interface for accessing the GDevices device driver.

A GDevice describes to Color QuickDraw the pixel arrangement the device uses. Pixel arrangement includes whether the device uses indexed or direct pixels, the color depth of the pixels, and the colors that the device can currently display. Along with the pixel arrangement the horizontal and vertical resolution of the device is stored in the GDevice.

The Color Manager of the Mac Toolbox uses GDevices to define what colors the GDevice is capable of displaying. If a color is re-

quested by the Color Manager that the GDevice is unable to display, the GDevice contains the information that enables the Color Manager to convert the desired color into one that the GDevice can display. The GDevice gives the programmer the ability to override the default color-mapping process.

The GDevice contains a reference to the device driver that is associated with either a display monitor or a printer. This device driver arbitrates access to hardware-specific features of the GDevice. Color QuickDraw uses the device driver of a GDevice to determine the location and addressing mode of the memory used by the display device.

Let's go over the specific fields of the GDevice. If you want further information on GDevice fields, I suggest reading the second edition of *Designing Card and Device Driver for the Macintosh Family.* This book is intended more for the programmer who needs to provide software for a display card. And yes, it is as boring as it sounds, but at this point it's the only halfway clear description of GDevices and their role that Apple has published.

- `gdRefNum` The gdRefNum contains the reference number for the device driver used by the GDevice. Offscreen-based GDevices don't have a device driver associated with itself.
- `gdID` Application-assignable identification for the GDevice. This identification tag is typically used by the applications overriding color-matching procedures.
- `gdType` The pixel type of the GDevice: direct, fixed, or indexed.
- `gdITable` A handle to the inverse color table used by the GDevice.
- `gdResPref` The resolution of the inverse color table.
- `gdSearchProc` A pointer to a list of search procedures registered with the GDevice.
- `gdCompProc` A pointer to a list of complement procedures registered with the GDevice.
- `gdFlags` The attribute flags for the GDevice.

- `gdPMap` The gdPMap field contains a handle to the pixel map that holds the dimensions of the pixel image, pixel resolution, color depth, and the color table for the GDevice.
- `gdRefCon` The gdRefCon is unlike other refcons in the Mac; this one is used to pass device-related values. Don't store any information in this field of the GDevice.
- `gdNextGD` The gdNextGD is a handle to another GDevice. The Mac uses this field to build a singly linked list of GDevices. If the GDevice is at the end of the list, this field will be set to nil. A GDevice that is associated with an offscreen buffer will not be included in the linked list, and this field will also be set to nil.
- `gdRect` The gdRect defines the boundaries of the GDevice.
- `gdMode` The gdMode field tells the driver how to set the device's mode.
- `gdCCBytes`, `gdCCDepth`, `gdCCXData`, `gdCCXMask` All four fields hold information for managing the mouse cursor on color screens.
- `gdReserved` Apple's placeholder for future growth; make sure this field is set to zero.

GDevice Potpourri

Working in conjunction with GDevices are several interrelated data structures. Some of these data structures are as important as a GDevice itself. Let's go over them.

Private GDevices

A GDevice is either a public GDevice or a private GDevice. A public GDevice is one that is available to every piece of code running on the Mac. The GDevice that represents your monitor is a public GDevice. A private GDevice is a GDevice that only your code has access to. Usually if your program creates a GDevice, you are creating a private GDevice. The GDevices that you will be creating in your game

programming will be private. In game programming the only public GDevice you'll run into is the display monitor's GDevice.

GDevice List

Every public GDevice that is created is added to a list of GDevices maintained by the Macintosh Toolbox. This list has the original name of GDevice list. Access to this list should only be through the functions `GetDeviceList`, which returns a handle to the first GDevice in the list, or `GetNextDevice`, which is passed a handle to a GDevice and will return a handle to the GDevice following the passed one in the list. Using the combination of these two traps, you can iterate through all the public GDevices. Make sure when you create a private GDevice that you do not add it to the GDevice list.

Main GDevice

The public GDevice that represents the display monitor showing you the menu bar is referred to as the Main GDevice. Just as there can be only one menu bar on a Mac, there can be only one Main GDevice. Besides being the holder of the menu bar, the Main GDevices establishes the origin (location 0,0) for the global coordinate space used by QuickDraw. The global origin is located at the upper left-hand corner of the Main GDevice. All other public GDevice monitors are located relative to the Main GDevice and its established origin.

Current GDevice

The most important global data structure used with GDevices is the current GDevice. Whenever Color QuickDraw draws, it always draws within the context of the current GDevice. When you create a private GDevice, you must make it the current GDevice to insure that QuickDraw will use your GDevice. You can retrieve a handle to the current GDevice by calling the function `GetGDevice()`. To make your GDevice the current GDevice, pass a handle to your GDevice as the parameter to the function `SetGDevice()`.

Color Graphics Port

Once you have created a PixMap and a GDevice, you still need to create a grafPort for QuickDraw to render into. Since you'll be creating your port in a color environment, an old-fashioned black-and-white grafPort will not do. You'll need the new and improved color graphics port or CGrafPort. Here is a CGrafPort in its full C structure glory.

```
typedef struct CGrafPort {
      short              device;
      PixMapHandle       portPixMap;
      short              portVersion;
      Handle             grafVars;
      short              chExtra;
      short              pnLocHFrac;
      Rect               portRect;
      RgnHandle          visRgn;
      RgnHandle          clipRgn;
      PixPatHandle       bkPixPat;
      RGBColor           rgbFgColor;
      RGBColor           rgbBkColor;
      Point              pnLoc;
      Point              pnSize;
      short              pnMode;
      PixPatHandle       pnPixPat;
      PixPatHandle       fillPixPat;
      short              pnVis;
      short              txFont;
      Style              txFace;
      char               filler;
      short              txMode;
      short              txSize;
      Fixed              spExtra;
      long               fgColor;
      long               bkColor;
      short              colrBit;
      short              patStretch;
      Handle             picSave;
      Handle             rgnSave;
      Handle             polySave;
      CQDProcsPtr        grafProcs;

} CGrafPort, * CGrafPtr;
```

Looking Backward

When building Color QuickDraw, Apple needed to make sure that existing black-and-white-based applications still worked on color Macintoshes. Color graphic ports were created as part of this backward-compatibility goal. In this creation Apple made sure that all the necessary fields of the original grafPort maintain their type and location within the structure. This magic is accomplished by either reinterpreting the existing fields or overlaying new fields while keeping the CGrafPort structure the same size as the original grafPort. The main difference is the lack of a bitmap field. The bitmap has been replaced by a handle to a pixmap. When building a color offscreen you will have to build a pixmap and then connect it up to your CGrafPort. Color QuickDraw can distinguish between a grafPort and CGrafPort by examining the portVersion field of the CGrafPort. If the two most significant bits are set, then the port is a CGrafPort. The portVersion field of a CGrafPort occupies the same offset as the rowBytes field of the grafPort bitmap structure. And since black-and-white QuickDraw required that the rowBytes field have the two uppermost bits cleared, this provided a quick and dirty method of determining which port type was created. You know why God could create Heaven and Earth in six days . . . no backward compatibility.

Examining CGrafPorts

You'll be creating a bevy of color graphic ports in your game programming, so let's break out each field of the structure that differs from the grafPorts and examine it.

- `portPixMap` A handle to the port's pixel map into which Color QuickDraw will image. On opening a CGrafPort the portPixMap handle will be referencing the pixel map of the main device.
- `portVersion` The two uppermost bits will be set to flag that the port is a color port; the rest of the field holds the version number of Color QuickDraw that created the color port.

- `chExtra`, `pnLocHFrac`, `grafVars` These three fields aren't of much importance in game programming. Check out *Inside Mac*, volume V, if truly interested.
- `portRect` Does the exact same thing as the portRect field did for a black-and-white grafPort. It defines the boundaries and coordinate space for the color port.
- `bkPixPat` Whenever Color QuickDraw needs to update the background of the port, it will use a pixel pattern referenced by the handle of this field.
- `rgbFgColor` Holds the requested RGB color you want Color QuickDraw to draw with.
- `rgbBkColor` Holds the requested RGB color you want Color QuickDraw to erase with.
- `fillPixPat` Holds a handle to a pixel pattern for filling. This field is used internally by Color QuickDraw.
- `fgColor` Where rgbFgColor holds the RGB color you requested, fgColor contains the pixel value Color QuickDraw will actually use for drawing.
- `bkColor` Same situation as fgColor but as applied to background colors.

The remaining fields aren't of much interest to you in game programming. You've been introduced to the main player involved in color offscreens for pixel depths of 8 bits and less. Before we plow ahead and start building color offscreens, let's spend a few pages going over bit depths greater than 8.

Tons o' Colors

In 1989 Color QuickDraw celebrated its second birthday, maturing with the introduction of 32-Bit QuickDraw. 32-Bit QuickDraw, or TrueColor QuickDraw, as Apple's marketing people attempted to

have us call it, enabled Color QuickDraw to have more than 256 colors on the screen at once. A lot more than 256 colors, like around 16,000,000 colors.

32-Bit QuickDraw was originally released as a system extension for system 6 and has been included as part of all system software since system 7. 32-Bit QuickDraw included a giant bag of enhancements and changes for Color QuickDraw, which were first documented in *Inside Mac*, volume VI (the volume that's bigger than the Manhattan white pages). We'll be going over just the parts of 32-Bit QuickDraw that affect color offscreens.

Direct Pixels

All pixels that you have dealt with to this point have been indirect pixels. Indirect, since the pixel value does not hold the actual RGB color value, but only points to it. With 32-Bit QuickDraw you can now have pixel values that are the actual color and not just a pointer. These types of pixels are called direct pixels.

32-Bit QuickDraw allows two sizes of direct pixels: 16-bit pixels and 32-bit pixels. With 16-bit pixels you can see 32,767 colors at once, and with 32-bit pixels Color QuickDraw can display 16,777,216 colors. A direct pixel is a scaled RGB color, where only so many bits of each color component of the full RGB color structure are used. In the case of 16-bit direct pixels, each color component is assigned 5 bits. Each component of red, green, and blue can hold a value from 0 to 32. With only 5 bits used for each color component, a leftover bit—the most significant bit of the word—is not used and is marked as reserved. A 32-bit direct pixel uses 8 bits for each color component, so that each component of red, green, and blue can hold a value from 0 to 255. By assigning 8 bits per color component, a 32-bit direct pixel uses only 24 bits to describe a color. The remaining 8 bits (the most significant byte of the long word) is marked as reserved, the same as the remaining 1 bit of a 16-bit direct pixel.

In programming with direct pixels you might see the unused bits at the top of the pixel referred to as the alpha channel pixels. An alpha channel is a technique in computer graphics that allows graphic images to have levels of transparency. With 32-bit pixels you could have 256 levels of transparency per pixel, where a value of 0 is a fully opaque pixel and a value 255 gives a pixel that is fully transparent. Some video cards for the Mac will display the levels of transparency for the use of combining the Mac signal with external video source. Another use of alpha channels can be found in high-end game consoles like 3DO, whose systems can combine one chunk of pixels with a background image using the alpha channel. By using the alpha channel these systems can produce effects such as semitransparent game elements. An alpha channel is sometimes referred to as a linear key by programmers with more of a video than a computer background.

2 Bits, 24 Bits, 32 Bits, a Dollar

32-Bit QuickDraw introduced some great features, and along with those features came some interesting problems. The main one was how to put all those pixels on a video card. A video card with the dimensions 640 ˘ 480 pixels with a depth of 32-bit pixels will need 1,228,800 bytes of memory on board the video card. For QuickDraw to render within this card, it needs to have memory access to all the bytes on the video card. Herein lies the problem. The Mac addressing scheme before 32-bit QuickDraw was a 24-bit addressing system, which gave the Macintosh access to 16 megabytes of memory. In this memory map each NuBus card is allocated a 1-megabyte slice of the memory-map pie. A 1-megabyte slice is not large enough to hold a full 32-bit deep frame buffer. To overcome this situation, 32-Bit QuickDraw gave the Mac the ability to switch its memory mode from 24-bit to 32-bit access. With a range of 32 bits the Mac can now address a memory map of 4 gigabytes of memory. Each NuBus card now gets a slice of 256 megabytes, more than enough to hold a 32-bit deep frame buffer. Whenever 32-Bit QuickDraw needs to draw in a 32-bit frame buffer, the addressing mode is temporarily switched from 24-bit to 32-bit addressing. After the drawing operation the memory mode is set back to 24-bit address-

ing. You can do the same thing by using the Toolbox function `SwapMMUMode`.

The above description assumed that the Mac was using the 24-bit memory mode that was available in 1989. In the modern times we live in you can set your Macintosh to use 32-bit memory access at all times. If this is the case, then `SwapMMUMode` will return the memory mode the Mac was in before it was switched into 32-bit mode. Forgetting to set the memory mode to 32-bit access before using a NuBus card and forgetting to restore the original memory mode after changing it are both sources of many crashes in game programming—a good tip to remember in your future game programming adventures.

Let's Build Some Offscreens

All right, enough is enough. Let's stop messing with the bit players of color offscreens and get down to the real business of building some.

Color Does Not Include Black and White

The library you will build is designed to work with any Mac model that includes Color QuickDraw. This requirement includes all Macs built since 1987, except the original Mac portable (Apple followed the television industry naming convention here—if you can put a handle on it, you can call it portable). This demand for Color QuickDraw still leaves several million Macs that this library will work correctly on.

All public functions in the offscreen library are passed a parameter of type `OffscreenHdl`. This handle to the structure of type `OffscreenStruct` contains all the information an offscreen needs to do its job. The library includes functions to build, destroy, and draw with these `OffscreenHdls`. Here is the structure definition for the offscreen handle.

```
typedef struct OffscreenStruct {
      short             depth;
      Rect              bounds;
      CGrafPort         offPort;
      GDHandle          offDevice;
      PixMapHandle      offPixels;
} OffscreenStruct, * OffscreenPtr, ** OffscreenHdl;
```

Make an Offscreen

To build a color offscreen you'll call the function `CreateOffscreen`. This function will return an error code and build you an `OffscreenHdl`. `CreateOffscreen` uses the following prototype.

```
OSErr CreateOffscreen( short            depth,
                       Rect *           globalBounds,
                       CTabHandle       startColors,
                       OffscreenHdl *   offscreen);
```

The depth you pass `CreateOffscreen` can be any of the following pixel depths: 1, 2, 4, 8, 16, or 32 bits per pixel. If you pass in 0 as a wanted depth, `CreateOffscreen` will create an offscreen that is the depth of the monitor that intersects the rectangle you pass in as the parameter `globalbounds`. The rectangle that you will give to `CreateOffscreen` will be in global coordinates. Having the rectangle expressed in global coordinates allows us to be independent of any port's coordinates and easily find which monitor our offscreen intersects. The color table passed in will be copied and attached to the pixel map that `CreateOffscreen` will create. For requested pixel depths of 16 bits or greater, the color table parameter is unnecessary and can safely be set to nil. If you set the color table to nil for color depths of 8 bits or less, `CreateOffscreen` will use the color table that matches the requested depths that are included in the Mac ROMs. `CreateOffscreen` will return an error code as a result. If the result indicates that no error occurred, you will have created a living, breathing offscreen. Enough preamble. Let's look at some code.

```
OSErr CreateOffscreen( short            depth,
                       Rect *           globalBounds,
                       CTabHandle       startColors,
                       OffscreenHdl *   offscreen)
{
      OSErr            err;
      CGrafPtr         port;
      CTabHandle       colorTable;
      GDHandle         gDevice;
      short            buildDepth;
      GrafPtr          oldPort;
      PixMapHandle     pixMap;

      // Initialize the temporary variables
      err = noErr;
      port = nil;
      colorTable = nil;
      gDevice = nil;
      pixMap = nil;
      *offscreen = nil;
      buildDepth = 8;

// Find out at what depth the offscreen will be created at

      if(depth)
      {
            // First verify that the requested depth
                  // is supported
            if(depth == 1 || depth == 2 || depth = 4 ||
                  depth == 8 || depth == 16 || depth == 32)
                  buildDepth = depth;
            else
                  return(paramErr);

      }
      else
      {
            GDHandle    gd;

      // Find the GDevice that intersects the rectangle
            gd = GetMaxDevice(globalBounds);
            if(gd)
                  gd = (*(*gd)->gdPMap)->pixelSize;
            else
                  return(paramErr);
      }
```

First, check that requested depth is realistic. If the requested depth isn't valid, a generic parameter error is returned. If the requested depth is zero, you ask the Mac, through `GetMaxDevice`, to find the screen with the biggest pixel depth that intersects the offscreen's bounds rectangle. If `GetMaxDevice` does not return an intersecting screen, you respond with another parameter error.

Make a Color Port

After determining what pixel depth your offscreen will be, you need to create a color graphics port for your offscreen.

```
// Make a color graphics port for our offscreen

// First save the current port for later restoration
GetPort(&oldPort);

//Allocate a chunk of memory for our future port
port = (CGrafPtr) NewPtrClear( sizeof(CGrafPtr));
error = MemError();

if(port)
{
      //OpenCPort indirectly changes the current port
      OpenCPort(port);

      //Set the port to coincide with the offscreen's rect
      port->portRect = *globalsBounds;
      //Cut down the visRgn
      RectRgn( port->visRgn, globalsBounds);
      //Do the same with the clipping region
      ClipRect(globalBounds);

      // Just in case we can't make a color table later
      (*(port->portPixMap))->pmTable = nil;
```

Creating a color port for the offscreen is fairly simple. Allocate enough memory for a color port through `NewPtrClear`. Use `OpenCPort` to initialize and open the port. After `OpenCPort` the current port will be your offscreen's port. `OpenCPort` directs the port to use the pixels owned by the main screen and sizes the port to match that screen.

Once the port has been opened cleanly, it is sized to match the offscreen's bounds rectangle. The port's regions are chopped down to match the bounds of the port. If this step is skipped, future drawing in your offscreen may be incorrectly clipped.

As a precaution for a failure in creating a color table for your offscreen, you should zero out the color table of the port's pixel map.

```
// Just in case we can't make a color table later
(*(port->portPixMap))->pmTable = nil;

// Create or copy a color table for the offscreen

// Only make a copy of the table if the pixmap
// contains indirect pixels. If the pixmap contains
// direct pixels create a dummy color table to satisfy
// code that requires a color table in a pixmap.

if(buildDepth <= 8)
{
      if(startColors )
      {
            SignedByte  state;

            // Make sure the color don't fade away
            state = HGetState( (Handle) startColors);
            HNoPurge( (Handle) startColors);

            colorTable = startColors;
            err = HandToHand(&colorTable);

            // Restore table to its previous state
            HSetState( (Handle) startColors, state);
      }
      else
      {
            colorTable = GetCTable(buildDepth > 1 ?
                              64 + buildDepth : 33);
            err = colorTable ? noErr : resNotFound;
      }

}
else
{
```

```
        colorTable = (CTabHandle) NewHandle(
                                    sizeof(ColorTable) -
                                    sizeof(CSpecArray));
        err = MemError();
}

// On failure clean up and return error
if(err != noErr)
{
        SetPort(oldPort);
        CloseCPort(port);
        DisposePtr(port);
        return err;
}

// Create a pixmap for the offscreen
err = CreatePixMap(buildDepth, colorTable,
                           globalBounds,
                           port->portPixMap);
if(err == noErr)
{
        // Copy the ports pixMap reference
        pixMap = port->portPixMap;

        // Erase the pixels in the pixel map
        EraseRect(&port->portRect);

        // Build the GDevice
        err = CreateGDevice(pixMap, &gDevice);
}
```

After the color port is properly created you'll need to allocate a pixel buffer for that port to draw within. Without a pixel map you have an offscreen that's not of much use, like a wallet with no money. To create a pixel map for your offscreen you will need to create a color table for that pixel map. In the example code a color table is made by copying a color table that was passed in to `CreateOffscreen` or a default color table matching the pixel depth of the offscreen is retrieved from the system.

Before copying the color table with a call to `HandToHand`, you will mark the source color table as nonpurgeable. Unless you mark the source color table it could be purged during the copying. The result will be a useless color table or a bus error. Best to avoid both of these by the simple call to `HNoPurge`.

If the source color table is nil, you should create a valid color table for the pixel map. This is easily done since the Mac has several color tables just lying around in its innards. To resurrect one of these color tables for your own use, call `GetCTable` with the magic resource id. The magic resource id for color tables is 64 plus the pixel depth. `GetCTable` will then return a color table that matches the pixel depth. By passing 32, instead of 64, plus the pixel depth, you will get a gray-scale color table. For the pixel depth of one you'll need a gray-scale color table containing only the colors black and white.

Creating or copying a full-color table for a direct pixel map is a waste of memory. Theoretically direct pixels do not require a color table to work correctly, but in reality some code still assumes that a pixel map needs a color table regardless of the pixel map's depth. To satisfy these regressive pieces of code, you need to create the smallest possible color table for your pixel map.

Make a Pixel Map

Now that you have made a color port and a color table, you have the pieces to build a pixel map for your offscreen. Since building a pixel map chews up a fair number of lines of code, I have declared a separate function to perform this task. `CreateOffscreen` calls this function to build the pixel map for your offscreen.

```
OSErr CreatePixMap(   short          depth,
                      CTabHandle     colorTable,
                      Rect *         bounds,
                      PixMapHandle   pixMap)
{
      OSErr err;
      Ptr   pixelsPtr;
      short width;
      short pmRowBytes;

      // Initialize the locals
      err = noErr;
      pixelsPtr = nil;
      width = bounds->right--bounds->left;
```

```
// figure out the rowBytes size for the pixmap
pmRowBytes = ((depth * (width + 31) / 32) * 4;

// Make sure the rowBytes is realistic
if(pmRowBytes > kMaxRowBytes)
      return paramErr;

// Allocate a pixel buffer
pixelsPtr = NewPtr((long) pmRowBytes *
            (long) width);
```

In order to allocate a memory buffer large enough for the off-screen, you first need to determine the number of bytes per row used by the pixel map. You remember this value, right—it's the `rowBytes` field. Who could ever forget the wonderful formula that produces the proper number for the `rowBytes` field? Last time you saw the calculation for the `rowBytes`, the goal was a number large enough to hold all the bits in a row and an even number. This is still the goal, except now you want the number to be divisible by four. Why? By making it divisible by four you are forcing the bytes to be long-word-aligned in memory. Color QuickDraw likes long-word alignment and rewards it with increased drawing speed. As part of the `rowBytes` calculation you also need to account for the bit depth being greater than one by multiplying by the pixel depth. That sums up the `rowBytes` formula.

After figuring out how wide the pixel map needs to be, you allocate a buffer wide enough and tall enough to hold all the pixels. Then you make sure the buffer is there, after which you can move on to the really exciting part, filling in all those pixel map fields. First you fill in the fields that are common to direct and indirect pixel maps.

```
// Allocate a pixel buffer
pixelsPtr = NewPtrClear((long) pmRowBytes *
            (long) width);

if(pixelsPtr)
{
      // Need to set all the non-zero fields
      (*pixMap)->bounds = *bounds;
```

```
(*pixMap)->rowBytes = pmRowBytes | 0x8000;
(*pixMap)->baseAddr = pixelsPtr;
(*pixMap)->hRes = 72 << 16;    // 72.0 ppi
(*pixMap)->vRes = 72 << 16;    // 72.0 ppi
(*pixMap)->pixelSize = depth;
(*pixMap)->pmVersion = 0;
(*pixMap)->packType = 0;
(*pixMap)->packSize = 0;
(*pixMap)->planeBytes = 0;
(*pixMap)->pmReserved = 0;
(*pixMap)->pmTable = colorTable;
```

- `bounds` Copy the offscreen's bounds to the pixel map's bounds.
- `rowBytes` Set the most significant bit of the `rowBytes` field to tell Color QuickDraw that it is dealing with a pixel map.
- `baseAddr` Copy the pointer to the pixel buffer you allocated earlier.
- `hRes, vRes` The resolution of an offscreen is usually 72 pixels per inch. Since `hRes` and `vRes` are fixed point numbers, you need to build a fixed-point version of 72.0. This is easily accomplished by shifting the integer portion (72) up into the high word of the resolution long word.
- `pixelSize` The pixel size of an indirect pixel map is equal to the requested pixel depth.
- `pmVersion` Set the pixel map version to zero.
- `packType` The pixel map you are building is not compressed or packed.
- `packSize` Since the pixel map is not compressed, set this field to zero.
- `planeBytes` The pixel map is not built from planes of color, so set the set the plane offset to zero.
- `pmReserved` For future compatibility, make sure you zero out this field.
- `pmTable` Assign the color table that was created for the pixel map.

Now that all the common fields have been assigned, you need to set the pixel map's fields that are specific to direct or indirect pixel maps. If the pixel depth is 8 bits or less, you have an indirect pixel map. We'll go over the indirect option first.

```
// Set the indirect vs. direct fields
            if(depth <= 8)
            {
                  (*pixMap)->pixelType = 0;
                  (*pixMap)->cmpCount = 1;
                  (*pixMap)->cmpSize = depth;
             }
```

- `pixelType` For indirect pixels this value should be zero.
- `cmpCount` The number of pixel components for an indirect pixel is one.
- `cmpSize` The size of each pixel component in an indirect pixel is equal to the pixel's depth.

Now let's go over the field settings for a direct pixel map.

```
      else
      {
              (*pixMap)->pixelType = RGBDirect;
              (*pixMap)->cmpCount = 3;
              (*pixMap)->cmpSize = depth == 16 ? 5 : 8;

              // Set the fields in the direct pixmaps
              // dummy color table
              (*colorTable)->ctSeed =
                                  3 * (*pixMap)->cmpSize;
              (*colorTable)->ctSize = 0;
              (*colorTable)->ctFlags = 0;
          }
      }
      else
           err =  MemError();

      // Return the functions result
      return err;
}
```

- `pixelType` Set the field to the constant RGBDirect to show that this pixel map holds direct pixels.
- `cmpCount` For direct pixels the component count is equal to three—one for each color component of red, green, and blue.
- `cmpSize` For direct pixels with a pixel depth of 16, each color component is made up of 5 pixels. For 32-bit-deep pixels, the component size is 8 pixels per each RGB component.

To clean up for a direct pixel map, you need to fix up the dummy color table that was built for use with your direct pixel map.

- `ctSeed` This can be set to any value. I picked what you see by copying what Apple suggested. If in doubt copy Apple.
- `ctSize` Show that the color table has no color entries by setting this field to zero.
- `ctFlags` Make sure this field is cleared.

To finish up, `CreatePixMap` returns the function's result back to `CreateOffscreen`. The pixel map just created will be attached to the color port that you created earlier. The only missing piece for building a color offscreen is a graphics device.

Make a Graphics Device

Once the pixel map has been created, you need to clear out the pixel map. The easiest way is to let QuickDraw clear it out for you. `EraseRect` will fill the offscreen with the current background color. Your offscreen has its port and pixel map; the only thing that is missing is the graphics device. A call to `CreateGDevice` will rectify this deficiency in your offscreen.

```
// Create a pixmap for the offscreen
err = CreatePixMap(buildDepth, colorTable,
                        globalBounds,
                        port->portPixMap);
if(err == noErr)
```

```
{
      // Copy the ports pixMap reference
      pixMap = port->portPixMap;

      // Erase the pixels in the pixel map
      EraseRect(&port->portRect);

      // Build the GDevice
      err = CreateGDevice(pixMap, kInvRes, &gDevice);

}
```

To build a graphics device you will need to give `CreateGDevice` a reference to the pixel map you just built. Like `CreatePixelMap`, `CreateGDevice` spends most of its time filling in the fields of the graphics device based on the pixel depth of the offscreen and building an inverse color table.

```
OSErr CreateGDevice( PixMapHandle pixMap,
                     short inverseRes,
                     Handle * resultGDevice)
{
      OSErr       err;
      GDHandle    aGDevice;
      ITabHandle  baseITab;

      // Initialize the locals
      err = noErr;
      aGDevice = nil;
      baseITab = nil;
      *resultGDevice = nil;

      // Create a handle for the GDevice
      aGDevice = (GDHandle) NewHandleClear(sizeof(GDevice));

      if(aGDevice )
      {
            baseITab  = (ITabHandle) NewHandleClear(2);
            if(!baseITab)
            {
                  DiposeHandle((Handle) aGDevice );
                  return MemError();
            }
```

```
// Start filling in the fields of the gDevice
(*aGDevice)->gdType =
(**pixMap)->pixelSize ? clutType : directType;
(*aGDevice)->gdITable = baseITab;
(*aGDevice)->gdResPref = inverseRes;
(*aGDevice)->gdPMap = pixMap;
(*aGDevice)->gdRect = (*pixMap)->bounds;
(*aGDevice)->gdMode = -1;
```

The buffer that will become your graphics device is zeroed out by allocating the handle with a call to `NewHandleClear` instead of calling `NewHandle`. Having the buffer cleared saves you the tedium of clearing all the fields directly.

A handle is created that holds only two bytes. This handle will later become the inverse color table for the pixel map. If the starter inverse table could not be created, you should dispose of the graphics device and bail from `CreateGDevice` (remembering to return the error result).

Let's go over the few fields in the graphic device you need to set to something other than zero.

- `gdType` Set whether the graphics device will be addressing direct or indirect pixels.
- `gdITable` This field holds the handle to an inverse color table.
- `gdResPref` Set this field to the inverse color table's resolution that is passed in as a parameter to `CreateGDevice`. The normal value passed is four.
- `gdPMap` Assign this field the pixel map used by your offscreen.
- `gdRect` Copy the bounds rectangle from the pixel map to this field.
- `gdMode` Make sure this field is -1 to alert Color QuickDraw that this graphic device has nothing to do with a display monitor.

```
            if((*pixMap)->pixelSize > 1)
                SetDeviceAttributes(aGDevice,
                                    gdDevType, TRUE);

      SetDeviceAttributes(aGDevice, noDriver, TRUE);

      if((*pixMap)->pixelSize <= 8)
      {
            MakeITable( (*pixMap)->pmTable,
                        (*aGDevice)->gdITable,
                        (*aGDevice)->gdResPref );

            err = QDError();
      }
}
else
      err = MemError();
```

If the offscreen is color, the graphic device flags field, `gdFlags`, needs to show that the graphic device is for a color offscreen. Additionally, since the graphic device is for an offscreen the `gdFlags` field needs to show that there is no driver associated with the graphic device. Both of these flag settings are made through the use of `SetDeviceAttributes`.

After all the fields and flags have been set, a check is made to see if the graphic device will be interfacing with indirect pixels. If so, you need to create an inverse color table for the graphic device to use. You need to pass to `MakeITable` a color table, the starter inverse table that was created earlier, and the requested resolution for the inverse table. `MakeITable` will then attempt to create an inverse color table for the graphic device. To verify that `MakeITable` was able to create the inverse table, a call to `QDError` is made.

Notice that a graphic device that uses direct pixels will end up with a reference to the two-byte inverse table that was created earlier. This does no harm, as long as you remember to dispose of the inverse table when you throw away the graphic device.

```
        // Check for any errors
        if(err)
        {
              if(baseITab)
                    DisposeHandle((Handle) baseITab );
              if(aGDevice)
                    DisposeHandle((Handle) aGDevice );
        }
        else
              *resultGDevice = aGDevice;
        return err;
}
```

Finally, before exiting the function you should check for any errors. If there was an error you need to dispose of any handles you created. On the fantastic occurrence that no errors were generated, return the handle to the graphic device you just created. The error result is also passed along to the calling function.

Finish Building the Offscreen

Now that the graphic port, pixel map, and graphic device have all been built, it's time to finish building the offscreen. A handle to the offscreen structure is allocated and filled in with the elements of the offscreen you created. At this point all you need to do is return the handle that holds the offscreen to the caller along with the error result.

```
if(!err)
{
      OffscreenHdl aOffScreen;

      aOffScreen = (OffscreenHdl)
                  NewHandle(sizeof(OffscreenStruct));
      if(aOffScreen)
      {
            (*aOffScreen)->depth = depth;
            (*aOffscreen)->bounds = *globalBounds;
            (*aOffscreen)->offPort = port;
            (*aOffscreen)->offDevice = gDevice;
            (*aOffscreen)->offPixels = pixMap;
```

```
                *offscreen = aOffscreen;
        }
        else
                err = MemError();
}
```

If an error was encountered while building the parts for the offscreen, you need to dispose of all data structures that were allocated from the heap. One detail on handling an error result is to make sure that you restore the current port setting before leaving the function. If you forget to do this, you will have disposed of the port QuickDraw thinks is the current port. On the next call to QuickDraw your day will be rudely interrupted by a visit from your friendly debugger.

```
        if(err)
        {
                // Restore the original port
                SetPort(oldPort);
                if(gDevice)
                {
                        DisposeHandle(
                                        (Handle)(*gDevice)->gdITable);
                        DisposeHandle((Handle)gDevice);
                }
                CloseCPort(port);
                DisposePtr(port);
                DisposeHandle( (Handle) colorTable);
        }
        return err
}
```

Destroying Offscreens

If there is a function that creates offscreens, there should be a function that destroys them. `DestroyOffscreen` will terminate the offscreen with extreme prejudice.

```
void  DestroyOffscreen( OffscreenHdl offscreen)
{
      ASSERT(offscreen);

      if((*offscreen)->offPort)
      {
            GrafPtr currPort;

            // Make sure that the current port isn't
            // our port

            GetPort(&currPort);
            if(currPort == (*offscreen)->offPort)
            {
                  GetCWMgrPort(&currPort);
                  SetPort(currPort);
            }

            // Kill off the pixel buffer
            DisposePtr(((*offscreen)->portPixMap)->baseAddr);

            // Kill off the color table
            if((**(*offscreen)->portPixMap).pmTable)
                  DisposeCTable(
                  (**(*offscreen)->portPixMap).pmTable);

            // Close and kill the port
            CloseCPort((*offscreen)->offPort);
            DisposePtr((Ptr) (*offscreen)->offPort);

      }
      if((*offscreen)->offDevice)
      {
            if(GetGDevice() == offDevice)
                  SetGDevice( GetMainDevice());

            (**(*offscreen)->offDevice).gdPMap = nil;
            DisposeGDevice((*offscreen)->offDevice);

      }
      DisposeHandle( (Handle)offscreen);
}
```

`DestroyOffscreen` disposes of all the hard work you went to in `CreateOffscreen`. If the offscreen has a color graphics port reference, the pixel buffer is destroyed followed by the retrieval and

destruction of the pixel map's color table. The wanton destruction is continued with the disposing of the color port. Before disposing of the port make sure you close the port with `CloseCPort`. If the offscreen has a handle to a graphic device, you'll need to free the graphic device and its inverse color table through `DisposGDevice`. The pixel map reference in the graphic device should be cleared out before your call to `DisposGDevice` just in case `DisposGDevice` gets ambitious and tries to dispose of the pixel map again.

One detail you need to take care of when disposing of an offscreen is to make sure that QuickDraw doesn't think that your offscreen is the current drawing environment. Failure to do this is a quick trip to Macsbug. `DestroyOffscreen` first checks that the current port is the same port you're about to destroy. If the two ports are one and the same, you should redirect QuickDraw to use another port. You can choose any port you want, but the easiest port is one that's guaranteed always to exist, like the window manager's port. The same fatal situation can arise if you dispose of the graphic device that QuickDraw is currently focused on. An equivalent test to see if the current graphic device is the same as the offscreen's graphic device is made. If they match you need to have QuickDraw focus on any graphic device other than the one you're about to wipe off the face of the earth. Again, choose any graphic device, but the best one is one you know will be there. Since every Mac has a main graphic device, telling QuickDraw to use the main device will always be a safe choice.

Using Color Offscreens

Now that you can create and destroy offscreens at will, let's build a program that will exercise these functions. On your disk you will find the sample program for this section of the book. Run the compiled sample application. To execute the offscreen demo select the demo item from the action menu. A graphic rainbow, reminiscent of a Grateful Dead T-shirt, will be drawn on screen. After following the on-screen prompt to click the mouse button, the screen will be

erased and the same rainbow will, after a delay, appear on-screen. The first rainbow was drawn on-screen using Color QuickDraw. The second rainbow is drawn using Color QuickDraw in an offscreen buffer and then transferred on-screen. To see where the action is, let's look at the function `DemoOffscreenLib`.

```
void DemoOffscreenLib(WindowPtr aWindow)
{
      OffscreenHdl      offscreen;
      Rect              globalRect;
      Point             offsetAmount = {0,0};
      OSErr             err;
      CTabHandle        colors;
      GDHandle          currDevice, offDevice;
      CGrafPtr          offPort;

      SetPort(aWindow);

      DwiddleBits(aWindow,
                  "On Screen Drawing (Click to continue)");

      while(!Button())
          ;
```

The function `DemoOffscreenLib` calls `DwiddleBits` to draw the rainbow on-screen. The demo then sits in a tight loop until the user hits the mouse button.

```
globalRect = aWindow->portRect;
GlobalToLocal(&offsetAmount);
OffsetRect(&globalRect, offsetAmount.h,
                        offsetAmount.v);

EraseRect(&aWindow->portRect);

currDevice = GetGDevice();

err = CreateOffscreen(8, &globalRect,
                  (**(*currDevice)->gdPMap).pmTable,
                  &offscreen);
```

Before an offscreen is created, the bounds rectangle of the on-screen window is converted to global coordinates. Before creating

the offscreen, the demo stashes away the current graphic device so that it can be restored when the demo is finished. Now with everything in place, the offscreen buffer is created into which the demo will draw. In creating the offscreen I passed the color table of the current graphic; this will insure that the offscreen has a color environment that matches the screen.

```
if(err == noErr)
{
      CGrafPtr    offPort;

      UseOffscreen(offscreen);
      DwiddleBits(aWindow, "Offscreen Drawing");
      ReleaseOffscreen(offscreen);
}
```

Making sure that there were no errors in creating the offscreen, the demo then calls the function `UseOffscreen`, a function of the offscreen library that you haven't seen before. The function `UseOffscreen` sets up QuickDraw to image into your offscreen instead of the monitor's screen. Before you change the focus of QuickDraw to one of your offscreens, you'll want to remember the graphic port and device that QuickDraw is currently using. Guess what? `UseOffscreen` does exactly that.

```
void  UseOffscreen(OffscreenHdl offscreen)
{
      ASSERT(offscreen);

      // Remember the current graphic environment
      GetPort( &(*offscreen)->prevPort);
      (*offscreen)->prevGDevice = GetGDevice();

      // Set our offscreen as
      // the current graphic environment

      ASSERT((*offscreen)->offPort);
      ASSERT((*offscreen)->offDevice);
      SetPort((GrafPtr) (*offscreen)->offPort);
      SetGDevice((*offscreen)->offDevice);
}
```

Now that QuickDraw is wearing graphic blinders and can only draw in the offscreen, `DwiddleBits` is called to generate a rainbow in the offscreen pixel buffer. On-screen you will see nothing, but offscreen QuickDraw is slamming pixels left and right. Offscreen drawing requires a certain element of trust that QuickDraw is really doing something back there.

> I've been throwing around the word *blit* quite a bit without the benefit of a definition. Sorry about that. Here you go, the full definition of blit.
>
> blit / *blit* / v. blitted, blitting, blits. To transfer a large matrix of bits from one location in computer memory to another; usually the memory being copied is being used to alter the screen's display ("The pixel image is blit from offscreen to on-screen")
>
> Derived from the PDP-10's Block Transfer instruction whose assembler mnemonic was BLT. Not to be confused with the 68000 Branch Less Than, BLT, instruction.
>
> blitter / *blit' r* / n. A dedicated piece of hardware or software used to perform pixel blits

After `DwiddleBits` is done re-creating the '60s, QuickDraw's graphic environment must be restored to its original state. The utility function `ReleaseOffscreen` will do exactly that. `ReleaseOffscreen` is the matching function for `UseOffscreen`. Whenever you make a call to `UseOffscreen`, make sure you have a matching call to `ReleaseOffscreen`. Your goal when using offscreens is to make no sounds, leave no footprints.

```
void  ReleaseOffscreen(OffscreenHdl offscreen)
{
      ASSERT(offscreen);

      // Set the saved graphic environment as the
      // current graphic environment.

      ASSERT((*offscreen)->prevPort);
      ASSERT((*offscreen)->prevGDevice);
      SetPort((GrafPtr) (*offscreen)->prevPort);
      SetGDevice((*offscreen)->prevGDevice);
}
```

Sitting somewhere in the demo program's memory is your rainbow. To make the offscreen graphic appear on-screen you'll use the Toolbox call `CopyBits`. We won't be going into any details concerning `CopyBits` at this point. We'll save that for later, where we'll go into so much detail it'll make your ears bleed. For right now, all you need to know is that `CopyBits` will transfer your offscreen pixels to your on-screen display. In order to make your pixels visible `CopyBits` needs to know where to find your offscreen buffer. Regrettably, `CopyBits` is ignorant of the offscreen structure you have created. In order to educate `CopyBits`, the offscreen library has a function that will return the offscreen's graphic port. This function goes by the original name of `GetOffscreenPort` and does just that, gets the port for the offscreen. With the offscreen's graphic port `CopyBits` can now copy all of the pixels from the offscreen to on-screen.

```
            offPort = GetOffscreenPort(offscreen);

            // Blit the offscreen to the window
            CopyBits(  &((GrafPtr)offPort)->portBits,
                              &((GrafPtr)aWindow)->portBits,
                              &aWindow->portRect,
                              &aWindow->portRect,
                              srcCopy, nil);
            DestroyOffscreen(offscreen);
      }
}
```

After blitting the rainbow on-screen you no longer need the offscreen. Get rid of it. One call to `DestroyOffscreen` will dispose of any unsightly remains or messy streaks left by `CreateOffscreen`.

Now that you've experienced the thrill of building your very own offscreens, let's go see how you can let the Mac make the job easier.

4

Modern Offscreens

A Brave New GWorld

You may have noticed in the preceding chapter that creating a Mac offscreen is not what you would call obvious. On the scale of obtuse things you can do on the Mac, offscreens ranks right up there with SCSI termination. Funny thing is that Apple seemed to understand this and in a very un-Apple way decided to fix the confusion over offscreens by writing better code instead of writing bigger *Inside Macs*. The improved offscreen support from Apple first appeared with the 32-Bit QuickDraw extension and has been included in every system version since 7.0. Apple combined all the graphic

port, graphic device, pixel map, and color table minutiae in one cute little structure they called Graphic Worlds, or GWorlds for short. GWorlds are amazing, they make your code simpler and yet enable it to run faster.

What the Heck Is a GWorld?

Before you can run amok with GWorlds you need to understand what the heck a GWorld is. Let's look at the Apple header QDOffscreen.h and see what the structure definition for a GWorld is.

```
/* Type definition of a GWorldPtr */
typedef CGrafPtr GWorldPtr;
```

Wow . . . now that's a useful declaration. Glad I looked that up. Apple views GWorlds as their private territory, so that you have no need to know what is in a GWorld. Just pretend a GWorldPtr is CGrafPtr, and nobody gets hurt. This data hiding prevents you and I from messing with GWorlds in a way that will blow up in later Apple system software. Hide the cookies and you remove the temptation to eat the cookies. You can assume from the previous chapters that a graphic world contains a graphic port, pixel map, and graphic device and is very similar to the offscreen library you built.

All manipulation of GWorlds is accomplished through Toolbox calls. The main functions you will use with GWorlds are

- `NewGWorld`
- `DisposeGWorld`
- `UpdateGWorld`

There are about a dozen lesser calls you will use when working with GWorlds, but these three are the main ones you'll be using. Let's go over each one and see what makes it tick.

NewGWorld

```
QDErr NewGWorld(    GWorldPtr *offscreenGWorld,
                    short PixelDepth,
                    const Rect *boundsRect,
                    CTabHandle cTable,
                    GDHandle aGDevice,
                    GWorldFlags flags);
```

A GWorld is created by calling `NewGWorld` and passing in the parameters describing how you would like your GWorld to look. The declaration is similar to the `CreateOffscreen` function in the offscreen library, and they should be similar since they do the same thing. You'll need to see how the parameters interact with each other to understand the call fully.

- `offscreenGWorld` This field is where the allocated and correctly filled-in GWorldPtr will be if the function executes cleanly.
- `PixelDepth` A GWorld can have pixels that are 1,2,4,8,16, or 32 bits deep. You can also pass in a pixel depth of 0, which should give you invisible pixels but doesn't. By passing a depth of zero you are asking `NewGWorld` to create an offscreen with a pixel depth that matches the deepest screen your GWorlds bounds rectangle intersects. This is handy since most GWorlds are created to shadow an on-screen window.
- `boundsRect` Establishes the size and the coordinate system of the GWorld.
- `cTable` The color table you want the GWorld to copy and use. If you pass a nil handle for the color table, a default color table will be used that matches the requested pixel depth. If the requested pixel depth is 0 this field is ignored and the color table of the intersecting screen will be used instead.
- `aGDevice` If you set the `noNewDevice` flag in the GWorldFlags, this field will be used as the graphic device for the GWorld. Most of the time you'll want to create a fresh graphic device for

your GWorld, so you can set this field to nil in those cases. Also if you request a pixel depth of zero this field will be ignored.

- `flags` The flags field can be set as a combination of four flags: `pixPurge`, `noNewDevice`, `keepLocal`, and `useTempMem`. If you set the `pixPurge` flag, your pixel image will be located in a purgeable handle. If memory gets tight, your pixel image will be purged from the heap. By setting `noNewDevice` a graphic device will not be created as part of your GWorld. If using `noNewDevice`, make sure to pass in a graphic device for your offscreen to use. If the `keepLocal` flag is set, the pixel image will be located in main memory and not in the onboard memory of a graphics card. By setting this flag you give up one of the major advantages of `NewGWorld`, which is its ability to use the memory of a graphics card instead of main memory. Many display cards can greatly increase the speed of QuickDraw operations when the graphic world is located in the display card's memory space. Having the graphic world memory sitting next to the display hardware also means that the pixels will not have to be pushed through the bus's bottleneck on their way to the screen, again improving the speed of QuickDraw. The `useTempMem` flag will ask `NewGWorld` to allocate the graphic world from memory outside of your application. This is handy if you need to create an offscreen that will only be used temporarily.

If `NewGWorld` is able to satisfy the memory demands of your graphic world request, you'll be presented with an a `noErr` result code and handed a brand-spanking-new GWorld to play with.

DisposeGWorld

```
void DisposeGWorld(GWorldPtr offscreenGWorld);
```

When you're done abusing your GWorld, you will need to call `DisposeGWorld` in order to release the memory allocated to the GWorld. After freeing the GWorld, make sure you don't use that GWorld reference again, otherwise you'll be knee-deep in bus errors.

UpdateGWorld

```
GWorldFlags UpdateGWorld(   GWorldPtr *offscreenGWorld,
                            short pixelDepth,
                            const Rect *boundsRect,
                            CTabHandle cTable,
                            GDHandle aGDevice,
                            GWorldFlags flags);
```

In order to change an existing GWorld you can make a call to `UpdateGWorld` with the new settings that you wish your GWorld to adapt to. The parameters for `UpdateGWorld` are the same as `NewGWorld`. Changes to your GWorld through `UpdateGWorld` will not destroy the image in the GWorld, although the image may be altered due to changes in pixel depth or color table changes.

`UpdateGWorld` accomplishes its task by duplicating your GWorld, making the changes to the original and then copying the pixels back to the original pixel image. When the pixel's depth or color table is changed a pixel copy will then be performed. Keeping this process in mind you can see that if you call `UpdateGWorld` you'll need enough memory not only for your GWorld but also for the scratch copy. This can add up to a lot of memory quickly, so be prepared for this call to fail occasionally.

What's the Catch?

Graphic Worlds give you a convenient wrapper that encapsulates GDevices, pixmaps, and color graphic ports all in one easy-to-use API. Along with the convenience, GWorlds are your entry into QuickDraw hardware accelerators. There's got to be a catch, right? Heinlein was accurate about that free-lunch thing. The cost for this lunch is lack of backward compatibility. The GWorld routines are only in either the 32-Bit QuickDraw system extension or System 7.0 and beyond. If you wish your games to run on systems older than these systems, you'll have to give up using GWorlds.

GWorld routines are well over five years old, and the majority of people who would play your games will have them in their

ROMs. Keeping this in mind, we'll be using GWorlds for the rest of the book.

Playing with GWorlds

Now that you can create, destroy, and adjust GWorlds at will, let's find something fun to do with them. How about animating something? Seems to fit this chapter. One of my favorite code animations on the Mac is the original About Box for Lightspeed C (which became Think C and then finally Symantec C++). In the Lightspeed C's About Box you appeared to be flying through a star field right out of *Star Wars*. After spending a few minutes being mesmerized by this effect, you would click in the About Box to close it. Before the About Box would close you would get a hyperspace effect like the one you see whenever the Millennium Falcon makes a jump to hyperspace. This was a great About Box. I used to stare at it while trying to determine why my code would not compile. (While Lightspeed C had a great About Box, it had truly lame error messages. Most of the time it would just say you had a "Syntax Error" and give you the thrill of guessing of the hundreds of subtle things you could have goofed up which one it did not wish to compile.)

To demonstrate GWorlds in action, I thought you'd like to recreate this great moment in About Box History. On your disk you'll find a project called "GWorld Fun." Run the built application. You'll see a dialog box with flying stars. To end the demo, click the mouse button. Before exiting, the program will jump to hyperspace (the effect is increased greatly if you make whooshing sound effects) and then quit.

GWorld Fun main()

This is a quick and dirty demo. Don't confuse it with a proper Mac program. Any error in the program results in the program's quitting. No alert, no beeps, it just quits. With the recriminations out of the way, let's tear into the code. Let's start with main().

The first thing we do is initialize the Mac Toolbox. Next a dialog window is created that is nicely centered on the main monitor. If the dialog window creation was successful, a GWorld is created that matches the size and pixel depth of the dialog window. To accomplish this, the requested pixel depth to the NewGWorld call is passed a zero. Which if you remember tells NewGWorld to create an offscreen that is of the size of the passed-in bounds and is of the pixel depth of the deepest monitor that the passed-in bounds intersects. For this to work correctly the bounds must be in global coordinates. The quickest way to get the dialog window's bounds into global coordinates is to grab the bounding rectangle of the content region of the window. This region is already in global coordinates and saves you the trouble of translating coordinate spaces. By passing nil for the color table, NewGWorld will use the color table of whatever device it uses for the offscreen. When using zero as the pixel depth, whatever GDevice you would pass in is promptly ignored by NewGWorld, so don't bother passing one. None of the NewGWorld flags are used.

```
void main(void)
{
     DialogPtr      demoWindow;

     InitToolbox();

     // Center the demo window
     HIG_PositionDialog( 'DLOG', 128);
     demoWindow = GetNewDialog(128, nil,(WindowPtr)-1L);

     if(demoWindow)
     {
            GWorldPtr    starWorld;
            QDErr        err;

            // Create a GWorld the size of the window and
            // then depth of the screen the window occupies.

            err = NewGWorld(&starWorld, 0,
                        &((**((WindowPeek)
                          demoWindow)->contRgn).rgnBBox),
                        nil, nil, 0);
```

If the GWorld was created without any problems, the demo will then copy the bounds of the dialog window into a globally accessible rectangle that the rest of the animation code uses. Then the star field is initialized by calling `BuildStars` (I love having function names that resemble godlike powers—BuildStars, BuildMoon, BuildGuilt). BuildStars just fills in the array of stars by giving each star an initial random position in X, Y, Z space.

```
if( err == noErr)
{
      long  time;

      // Set the bounds that the stars will
      // exist in.
      gStarBnds = demoWindow->portRect;

      // Create a sky full of stars
      BuildStars();

      // Prepare the GWorld for QuickDraw

      LockPixels(starWorld->portPixMap);
      SetGWorld(starWorld,nil);
```

Now that the stars have been created, the GWorld is prepared for future drawing by using `LockPixels`. Since you'll be using LockPixels frequently in your game programming, let's take a break from looking at the demo and examine LockPixels.

```
Boolean LockPixels(PixMapHandle pm);
```

When a pixel map is created the image buffer that the pixel map owns is pointed at by a handle, not a pointer, as with a bitmap. Before any drawing can take place in the pixel image, the handle to the pixel image must be converted to a pointer. This is exactly what LockPixels does. It locks the handle and then replaces the handle reference in the pixel map with the direct pointer to the pixel image. Failure to call LockPixels will cause QuickDraw to think the handle is a pointer. QuickDraw will dereference the handle only once and will be pointing at a master pointer block and not the

pixel image. QuickDraw will then spray pixels all over this very important Memory Manager structure. You might crash right away, if you're lucky. If you're not so lucky, you won't crash until some undetermined time in the future, and then you get to spend some quality time with your debugger. I've been to this movie and you won't like the ending. Always remember to call LockPixels before drawing in your GWorld.

If you look at the LockPixels declaration you'll see that it returns a Boolean result. This Boolean result comes into use if you declared that the pixel image was purgeable when you created your GWorld. If the pixels have been purged from memory, LockPixels will return FALSE. You'll need to reallocate the pixels by either recreating the GWorld or using UpdateGWorld. In the case of "GWorld Fun" the GWorld was not created with purgeable pixels, so the result from LockPixels can be safely ignored.

Now that the GWorld is prepared, let's move to impulse speed. The star field is kept moving until the mouse button is clicked. Each frame of the star field animation is built by calling `CycleThruStars`. CycleThruStars is called with a parameter of TRUE to make sure the previously drawn stars are erased. CycleThruStars draws all the stars into the offscreen area that was created with NewGWorld.

```
// Fly through the stars until the mouse is clicked
while(!Button())
{
      // Create the next frame of the star animation,
      // erasing all the previously drawn stars.
      CycleThruStars(TRUE);

      // Move the animation frame to the screen.
      CopyBits((BitMap *) (*(starWorld->portPixMap)),
            &(demoWindow)->portBits,
            &demoWindow->portRect,
            &demoWindow->portRect,
            srcCopy, nil);
}
```

After CycleThruStars has finished creating the next frame of the star field animation, CopyBits is called to move the offscreen anima-

tion frame to on-screen. Continuously repeating this cycle produces a nice illusion that would even make Industrial Light and Magic proud.

After the user presses the mouse button, "GWorld Fun" makes the jump to hyperspace or engages warp drive, depending on which sci-fi religion you belong to. The hyperspace jump animation uses the same code as the flying stars loop. The only difference is that CycleThruStars is called with a FALSE parameter. By passing FALSE, CycleThruStars does not erase the stars at their previous positions. This produces trails of stars that resemble the hyperspace effect from *Star Wars*.

```
time = TickCount() + kNumHyperTicks;
while(TickCount() < time)
{
      // Create the next frame of the star animation,
      // previously drawn stars are not erased
      //producing a hyper space/warp effect.
      CycleThruStars(FALSE);

      // Move the animation frame to the screen
      CopyBits((BitMap *) (*(starWorld->portPixMap)),
            &(demoWindow)->portBits,
            &demoWindow->portRect,
            &demoWindow->portRect,
            srcCopy, nil);
}
```

The hyperspace loop runs for a fixed length of time (four seconds) by setting the time in ticks. The loop is controlled by adding how many ticks the hyperspace animation should run and then cycling through the hyperspace frames until TickCount returns that the allotted amount of ticks have passed.

Now seems as good a time as any to explore just what a tick is and why TickCount wants to count them.

```
unsigned long TickCount(void);
```

TickCount returns the number of ticks that have passed since your Mac was turned on. A tick is equal to 1/60 of a second, or

16.66 milliseconds for the electrical engineers among us. TickCount gives game programmers an easy-to-use heartbeat that is precise enough timing for most tasks. To time events with greater precision than TickCount can provide, you'll need to look at the Time Manager. If you're requiring QuickTime for your game, you could use the clock component to provide a high-precision timer.

To clean up after itself "GWorld Fun" first resets the pixel map reference in the GWorld back to a handle by calling UnlockPixels. After you let the pixels wander around, you tear down the GWorld by calling DisposeGWorld. Finally, the dialog window is destroyed with a call to DisposeDialog, and "GWorld Fun" is now done.

```
                // Let the pixels wonder around
                UnlockPixels(starWorld->portPixMap);

                // Then free the graphic world
                DisposeGWorld(starWorld);
            }

            DisposeDialog(demoWindow);
        }
}
```

How Do the Stars Move?

I first learned how to do a star field in an article in *MacTutor* (now *MacTech*) by Mike Morton. The basic idea is that each star is represented by an x, y, and z coordinate, where the z axis is considered going into and coming out of the screen.

```
typedef struct      {
        short x;
        short y;
        short z;
} Star, * StarPtr;
```

The stars are animated by reducing the z value, effectively moving the star toward you. The 3D appearance of the star field is given by mapping the x, y, and z position of each star to a two-dimensional

position in the window. The mapping transformation includes the impact that perspective will have on the viewer. The perspective portion of the mapping is what provides the 3D-ness of the animation. The mapping is done for each star in the star field by calling `MapStarToScreen` with a pointer to the star to map. The star's on-screen position is returned as a QuickDraw Point.

```
Point MapStarToScreen(StarPtr star)
{
      Point result;

      result.h = star->x * kProjectDistance / star->z +
                                    gStarCenter.h;

      result.v = star->y * kProjectDistance / star->z +
                                    gStarCenter.v;

      return result;
}
```

The two equations provide the perspective 3D mapping. For more information on perspective mappings, check out any good reference on 3D graphics. The constant kProjectDistance can be thought of as the focal length of the camera the user is viewing the star field through. Try changing this constant to see its impact on the transformation.

Combining all of the above to produce an animation is the responsibility of CycleThruStars. If the caller has asked CycleThruStars to erase the previous frame, the QuickDraw pen is reset to its normal setting, then the entire offscreen is painted with black. The pen pattern is set to white in preparation for drawing all the stars.

```
void  CycleThruStars(Boolean  eraseStars)
{
      short i;

      // Erase all the existing stars in the star field
      if(eraseStars)
      {
            PenNormal();
            PaintRect(&gStarBnds);
```

```
            PenPat(&white);
        }

        for(i = 0; i < kMaxNumberStars; i++)
        {
            Point screenPosition;

            // Calculate where the star will move to
            MoveTheStar(&gStars[i]);

            // Now Draw the star in its new position
            screenPosition = MapStarToScreen(&gStars[i]);
            MoveTo (screenPosition.h, screenPosition.v);
            LineTo (screenPosition.h, screenPosition.v);
        }
}
```

The array of stars is indexed with each star's new position being calculated and then projected onto the window. The star is rendered in the offscreen by first positioning QuickDraw's pen to the star's position and then making a one-pixel white dot with a call to `LineTo`. LineTo is passed the same coordinates as the `MoveTo` call, which will produce a rectangle the size of the current pen. The pen was set to 1 ˇ 1 with the earlier call to PenNormal.

Morton's article showed how to produce the star field effect without resorting to offscreen buffering. In fact the offscreen isn't necessary to produce this effect at all. I only combined the star field with the GWorlds to provide something a little more fun than bouncing rectangles.

Bit Banging

For Mac game animation you need two things: offscreen pixel buffers and something to move those pixels around with. That something is referred to as a blitter.

A blitter's purpose is to copy, or blit, pixels from one area of memory to another place in memory. Usually the blitter takes pixels from the offscreen buffers and moves them to the screen. Blitting done rapidly enough produces animation.

A blitter is either a piece of code or a hunk of dedicated hardware. Whether it's hardware or software, the blitter does the same job—it move chunks of memory around. Hardware blitters just do the job a lot faster than software blitters.

A blitter is the workhorse of any arcade game. All elements that are moved or animated in an arcade game are done through the blitter. The number of elements a game can animate on-screen at once is tied directly to the horsepower of the blitter. The faster the blitter, the more pixels it can move. The pixels can result in either filling the screen with hundreds of tiny elements or just a few very large elements. The tradeoff is between size and quantity. A game like Lemmings requires a whole bunch of tiny elements animating at once, while a game like Street Fighter II only has a few elements, but those elements are huge compared to the diminutive lemmings. A blitter will be tuned to provide better performance for the size of elements the game will need to animate.

A blitter is responsible for how many and how large your game elements can be. Tied in with size and quantity is frame rate. How many frames per second your game animates at is a function of how fast your blitter can move the game elements to their location from offscreen to on-screen. A weak blitter combined with either too many elements or elements that are too large will result in a low frame rate. A low frame rate will result in jerky-looking animation that translates into a hard-to-play arcade game. A good blitter won't make for a great arcade game, but a bad blitter will help produce a mediocre game. This section's goal is to help you avoid making mediocre games and give you the first step in creating great games.

What Is Blitting?

All right, so a blitter is the heart of any arcade game you write. But what does it really do? A blitter copies a rectangular hunk of pixels from one area of memory to another rectangular area of memory. To see blitting in action on your Mac, grab a draggable window and move it to a new location on the same monitor. When you release the mouse button the pixel image on your screen that looks like your window will be blitted to its new location. You just saw a software blitter in action.

Now, you might be wondering what's the difference between blitting pixels around and a simple block move of memory that you would perform with a call like `BlockMove` or `memcpy`. In a standard block copy of memory the bytes copied are required to be sequential in memory. Visually the memory looks like one long line of bytes, with no gaps or jumps. A blitter copies rectangular chunks of memory, where the address of each pixel is not necessarily sequential to the previous or next pixel. This detail makes all the difference between a block copy and a simple blit of pixels.

A Simple Blitter

To illustrate a simple blitter, let's create one. Our blitter assumes that it's copying 8 bits per pixel and that the pixels are arranged in a chunky layout. This blitter is just for illustration purposes. Don't use this blitter at home, kids.

`BrainDeadBlit` takes two pointers to pixel maps. The first pixel map is where the source pixels are stored. The second PixMapPtr is indicating where the pixels will be blitted to. The third parameter is the rectangular bounds that will be cut (actually copied) from the source to destination. The final rectangle indicates where the pixels should be slammed. Even though the destination location is described as a rectangle, only its top-left coordinate is used.

Before the actual blitting is done, the base address of where the pixels lie in each pixel map is copied to local variables. The same is done with the pixel maps rowBytes field. Copying the data out of the pixel map saves the code from having to retrieve these values continuously inside the blitting loops.

After stashing away those values, the blitting starts. The first blitting loop works down from the top of the source rectangle to the bottom. Before the real blitting starts, you need a pointer to the first pixel of the current source row. This pointer is found by starting with the base address of the pixel image and finding how many bytes to add to get to the current row. The current row that the outer loop is on, times the number of bytes per row, added to the base address will give you the starting address of the row. Adding the left coordinate to the previous sum will get you to the address of the first pixel you want to blit for this row. The same calculation is done to find the address of the first pixel of the destination row. Once the start of the source and destination are found, the blitting starts. The innermost loop works from the left side of the source bounds to the right side, blitting each pixel in between.

```
void  BrainDeadBlit(  PixMapPtr   srcMap,
                      PixMapPtr   dstMap,
                      Rect *      srcR,
                      Rect *      dstR)
{
      char *      srcAddr;
      char *      dstAddr;
      short       srcRowbytes;
      short       dstRowbytes;
      short       row, pixel;
```

```
        // Extract the start of the pixels out of the pixmap
        srcAddr = srcMap->baseAddr;
        dstAddr = dstMap->baseAddr;

        // Extract the rowBytes out of the pixmap
        srcRowbytes = srcMap->rowBytes & 0x3FFF;
        dstRowbytes = dstMap->rowBytes & 0x3FFF;

        for(row = srcR->top; row <= srcR->bottom; row++)
        {
             char *       srcPtr;
             char *       dstPtr;

             // Calculate the start of each row
             srcPtr = srcAddr +
                         (row * srcRowbytes + srcR->left);

             dstPtr = dstAddr +
                         (row * dstRowbytes + dstR->left);

             for(  pixel = srcR->left;
                   pixel <= srcR->right;
                   pixel++)
             {
                   *dstPtr++ = *srcPtr++;
             }
        }
}
```

Blitting as presented here seems pretty simplistic, and it is. As long as you're willing to have a blitter that is as slow as MPW linking and that can only copy rectangular sections, then your blitter will be simple. The hard part of writing a blitter is writing a fast one.

The Mac's Built-in Blitter: CopyBits( )

The Mac doesn't have a hardware blitter built into it, but it does have one of the best general purpose software blitters ever. This built-in blitter can copy pixels from anywhere to anywhere (limited to memory locations). And the pixels can be stretched, shrunk, colorized, dithered, quantized, or masked while being

copied. Each of these effects can be done individually or, amazingly enough, all at once. And what is the name of this *wunderkind* function? CopyBits. Such a utilitarian name for such a cool chunk of code is a shame; it should be called the WhizzoBlitBanger or something that sounds like it has the fins of a '59 Caddie. When the original version of CopyBits was written, one bit was equal to one pixel; so copying bits was equal to copying pixels, and the name has stuck even though it copies more bytes than bits.

```
void  CopyBits( const BitMap *srcBits,
                const BitMap *dstBits,
                const Rect *srcRect,
                const Rect *dstRect,
                short mode,
                RgnHandle maskRgn);
```

The first two parameters to pass to CopyBits are pointers to the bitmaps where the pixels start and where you want them to go. In order to use pixel maps you'll need to cast a pointer to a PixMap as a pointer to a bitmap. CopyBits will look at the rowBytes field of the passed pointer to determine if you have passed it a pointer to a bitmap or one pointing at a pixel map.

The next two parameters describe the bounds of the pixels that CopyBits should blit. If the rectangles are the same size you'll get a straight pixel copy (at least the pixels won't change size); otherwise CopyBits will stretch or shrink the pixels to fit the destination bounds.

The mode parameter allows you to set the logical operation CopyBits will perform when transferring the pixels. Most of the time you'll pass the constants `srcCopy` as your mode of choice. This mode asks CopyBits to just copy the pixels and not do anything fancy. With other mode settings you can have CopyBits perform various logical operations that produce a resultant pixel that is a combination of the source and destination pixel. The resultant pixel will replace the destination pixel.

The final parameter can be a region handle that will be used as a filter for which pixels should be transferred. Pixels in the source that

lie within the region bounds will be copied. Pass nil to indicate to CopyBits that you wish to have all your pixels transferred without any clipping.

Several QuickDraw settings external to the parameters you pass control how CopyBits will move your pixels. If the destination pixel map is part of a graphics port, CopyBits will clip the pixels to the intersection of the content region and the visible region. By setting the foreground color and background color of the destination port to other than black and white, you can colorize your pixel map like you're Ted Turner.

What All Does CopyBits Do?

Looking at the prototype for CopyBits can fool you. It looks like a straightforward Toolbox call. Hah, CopyBits has options like a '59 Caddie has chrome. The only Toolbox call more mysterious is Munger, and at least it has the good sense to have a name that sounds strange. By changing just one parameter (heck, it doesn't even have to be a parameter, you can change something in the graphic port), you can get your pixels to do some strange and wonderful things. Occasionally those things are even what you expected. Enough complaining, let's go over the various incantations you can make with CopyBits.

Scaling

By carefully setting the ratio of the source rectangle and destination rectangle, you can have CopyBits scale your image as it copies it. CopyBits is capable of shrinking the image in one direction while at the same time stretching it in the other direction. When CopyBits scales the pixels it does so with no filtering, which results in the image becoming blocky when it is scaled up and looking pixelly when it is shrunk. This aliasing of the imaging during the transfer is fine for games and you shouldn't worry about it, but you shouldn't be using CopyBits as a replacement for the scale function in PhotoShop.

```
SetRect(&sourceRect, 0, 0, 16, 16);
SetRect(&destinationRect, 0, 0, 32, 32);

CopyBits(srcBitsPtr, &thePort->portBits,
            &sourceRect,
            &destinationRect,
            srcCopy, nil);
```

In this code snippet the destination rectangle is twice as large as the source. The resultant image will be enlarged twice its original size through the call to CopyBits.

Transfer Modes

By changing the mode parameter to CopyBits you can get your pixels to combine with the destination pixels in useful and sometimes very strange ways.

Classic QuickDraw defined the original eight transfer modes: srcCopy, srcOr, srcXor, srcBic, notSrcCopy, notSrcOr, notSrcXor, notSrcBic. Color QuickDraw added nine more transfer modes for working with color pixels; addOver, addPin, subOver, subPin, adMax, adMin, blend, transparent, hilite.

Each of the transfer modes is a logical operation that QuickDraw will perform on each pixel as it copies them. Of the original eight transfer modes, the ones used the most in game programming are srcCopy, srcOr, and srcXor. The rest are used in rare circumstances. The color modes are fairly esoteric and not used that much in game programming. For a good discussion of transfer modes and some really nice color plates, look at *Inside Mac*, volume VI.

Clipping

To clip your transferred pixel to some arbitrary shape other than a rectangle, you can pass a region describing the shape. The region can come from either the standard OpenRgn-CloseRgn construct or from the nifty BitMapToRegion function. The region is applied to the pixels after any scaling or stretching is performed, so don't expect the region to adjust to match the destination rectangle. If

that's what you want, you can use the MapRgn call before calling CopyBits.

Colorizing

When Color QuickDraw was introduced, the Mac Toolbox provided a neat hack that allowed for CopyBits to colorize black-and-white bitmaps as they were copied. To perform this Turneresque magic you set the foreground and background colors of the destination graphics port to the colors you would like the bitmap to be converted to. Black pixels in the source bitmap are changed to the foreground color, and likewise the white pixels are altered to match the background color. CopyBits performs the following mapping using the table indices of the colors, not the RGB values.

```
src = color table index of source pixel
fg = color table index of foreground color
bg = color table index of background color

FinalPixelValue = (~(src) & bg) | (src & fg)
```

The mapping that CopyBits performs indicates several restrictions on what can be colorized. Your source bit map must be black and white; shades of gray don't count. The first and last entry in the source color table must be black and white, respectively. Your source and destination must be using indexed colors. Failure to meet these prerequisites will result in some very strange results, neither predictable nor useful. In game programming the colorizing is rarely used intentionally. Usually it's the result of forgetting to restore the foreground and background colors before performing a CopyBits.

Depth Conversion

If the pixel depth of the source pixel map doesn't match the destination pixel map's depth, CopyBits will convert each pixel from the source depth to the proper depth in destination. While in normal application programming this feature of CopyBits is convenient, in

game programming you'll go to great lengths to insure that CopyBits is not put in the position of having to perform a depth conversion.

Dithering

Along with 32-Bit QuickDraw's other goodies, Apple created the `ditherCopy` transfer mode. This transfer mode can be added to any other mode you are using with CopyBits: srcCopy + ditherCopy. With the ditherCopy transfer addition enabled, CopyBits will dither (or error-diffuse, for you graphic heads) the source pixel onto the background. By using dithering you can get more apparent color depth than the destination truly has. Like depth conversion, dithering is a useful adjunct for application programs but not that useful for game programming because of its need to chew on the pixels for extended periods of time.

The one reason some games use the dither mode is to achieve a cheap alternative to anti-aliasing text. If you image the text in an offscreen that is four times larger than the destination and render the text at four times the wanted size, you can then CopyBits the text on-screen using ditherCopy and get a good approximation of anti-aliased (otherwise know as *fuzzy*) text. This trick is good for making professional-looking About Boxes.

CopyBits Minutiae

Before we move on, let's look at a few things CopyBits does that might not seem obvious but are useful to know about. One of the neater things CopyBits does that I've always found impressive is it allows you to have the source and destination rectangles of the blit overlap. To appreciate this feature take a few hours or weeks and try to write a fast blitter that allows overlapping source and destination. If you're ever feeling really impressed with yourself as a programmer, give it a shot. It'll give you a new appreciation of the programmers at Apple.

CopyBits is also capable of blitting with one call a pixel image that spans across several monitors. This can be real handy and another feature of CopyBits that would be an effort to duplicate.

The last bit of minutiae about CopyBits is that it is the one call that any QuickDraw accelerator card is guaranteed to speed up. CopyBits is used by so many programs that by accelerating this one call an accelerator board will improve almost any program on the Mac. The impact this has on Mac game programming is that if a system has a QuickDraw accelerator, using CopyBits will almost always be faster than any blitter you could write. No matter how good a programmer you are, it's hard to beat well-designed hardware.

What Doesn't CopyBits Do?

If you've ever spent any time talking to Mac game programmers, you could get the idea that CopyBits is a barely able to run, let alone be the main engine for an arcade game. With all the knowledge about CopyBits you now possess, you would chalk up all this CopyBits bashing as a case of NIH syndrome (NIH: *not invented here*, a derisive term applied to all software not invented by you or your company, on the idea that if you didn't write the code, then the code must be inferior). Before you go defending the good code from Cupertino, you need to step back and see that for games involving screenfuls of animation CopyBits might not be the best choice as an animation engine. Heresy, say you. Not, say I. CopyBits, in its never-ending drive to be the Vegematic™ of blitters, gives up speed for generality.

When programming Mac arcade games it's accepted form to force the user into using a fixed number of colors and only one monitor. By enforcing these restrictions on the user, the programmer gains a known graphic environment. It's this foreknowledge that lets you easily write a blitter that is faster than CopyBits. CopyBits won't be aware of the restrictions you've placed on the user, and it has to perform a multitude of checks before it finally begins slamming pixels around. Your blitter on the other hand can skip all those checks and start blasting bits right away.

The speed advantage your blitter will have is that you get to start blitting while CopyBits is still figuring out what and where to blit. Once CopyBits has these details in hand it can kick into high gear. Given this insight into CopyBits, you can see that it would

probably be roughly the same speed as your blitter for moving large areas of pixels. But CopyBits' initial overhead makes it a bad choice for moving a large number of small areas of pixels. Now what's in most Mac arcade games? Usually lots of areas of small pixels.

However, before you leap into writing your own blitter for your game, make sure that CopyBits is not cut out for the job. No point in reinventing the wheel if you can borrow your neighbor's.

Who Is That Masked Blitter?

A straight blitter like CopyBits only copies rectangular chunks of pixels. Now, this limitation is all right if your game consists of nothing but animating bricks. But for most games you'll be animating elements that are more complex than squares. The technique for copying nonrectangular pixel areas is referred to as masked blitting.

Why Mask a Blit?

Using a mask while blitting allows you to place a pixel image on-screen without destroying the background. Using CopyBits to transfer the pixel image without some form of masking will result in part of the background being obliterated (see Figure 4-1). To rid yourself of this unsightly background destruction, you need to implement one of the several methods of masking your blits.

Figure 4-1. CopyBits() with a mask

If you've ever watched one of those *Making of Star Wars/Star Trek/Superman* or any special-effects extravaganza, you have seen

masking in action. Anytime the Starship *Enterprise* flies, its flight is assisted by a mask, or a flying matte, as it's called in these specials. The mask is used to mask out the area of the background where the Starship will be dropped in place. The same general idea is used with a blitter. The mask is a data structure that tells the blitter which bits of the source image should be copied and which pixels to ignore. Generally, the mask for a blitter is a 1-bit-deep shadow of the pixel image where the black pixels represent the pixels of the source image that the blitter needs to copy. Anywhere there is a white pixel in the mask, the blitter will not transfer the corresponding source pixel (see Figure 4-2).

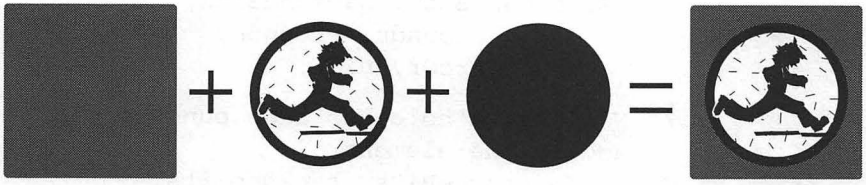


Figure 4-2. A pixel blit using a mask

Masking not only stops you from destroying the background but also enables you to have several overlapping elements on the screen at one time. There are very few games you could design that would avoid the elements' overlapping one another at some point. And even if you could design a game without overlapping shapes, it would probably be boring. Tetris, of course, is the exception that proves the rule. Using a mask also allows your game elements to be any conceivable shape. Even ones with holes. With one mask you can create a game element that can preserve the background, overlap other shapes politely, and appear transparent in areas (see Figure 4-3).

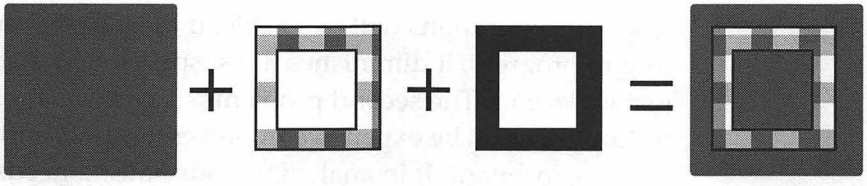


Figure 4-3. Blitting a source image with a hole

Masking Math

The traditional method of implementing a masked blitter is to have each pixel in the source image perform the following logical operations. First, the mask is used to punch a hole in the background where the source image will go, using the logical Or operator. Then the source image is combined with the mask/background combination through the logical And operator. The result is a masked pixel on-screen.

```
//    source & (mask | background)

// First mask or'ed with the background
CopyBits(maskBits, backgndBits,
            &bounds, &bounds,
            srcOr, nil);

// Now that a hole has been punched out
// add in the element
CopyBits(sourceBits, backgndBits,
            &bounds, &bounds,
            srcAnd, nil);
```

CopyBits can perform a masked blit by breaking up the blit into two steps and taking advantage of the transfer modes. Beyond the two steps and some source pixels to copy, you will need a bitmap that provides the matching mask. Following the magic formula, the mask bitmap is combined with the background using CopyBits' srcOr transfer mode. After the hole has been punched out with the mask, the source pixels are cut into place with the srcAnd transfer mode.

Using CopyBits in a two-step process for your game blitter has a couple of holes in it. The first one is that if CopyBits' blit is performed on-screen with large enough pixel areas, the two separate CopyBits operations will be visible. If the player can see your blitting in progress, it diminishes the suspension of disbelief. Plus, it just looks bad. The second problem is a performance issue. One call to CopyBits can be expensive in processor time, and two such calls can be exorbitant. If in analyzing your blitting needs neither of these hindrances is an obstacle, then go ahead and use CopyBits

and forget about using a custom blitter. You don't need a chain saw just to cut butter.

Luckily, if it does look like you need a chain saw, the Mac Toolbox has a few buried in the ROM that you can test out before building your own.

CopyMask

Within the Mac ROM is a simple blitter that supports using a separate bitmap/pixmap to mask the pixel copying. The Mac function `CopyMask` was introduced in the Mac Plus and System 3.something.

```
void CopyMask(const BitMap *  srcBits,
              const BitMap *  maskBits,
              const BitMap *  dstBits,
              const Rect *    srcRect,
              const Rect *    maskRect,
              const Rect *    dstRect);
```

At the time of `CopyMask`'s introduction, Color QuickDraw was just a gleam in some Apple programmer's eye. That meant bitmaps were the only pixel structures around at the time to be copied and masked. With the introduction of Color QuickDraw, `CopyMask` was altered to work with color pixmaps.

As with `CopyBits`, `CopyMask` takes a source and destination bitmap pointer. It can be a real bitmap pointer, a pixel map pointer masquerading as a bitmap pointer, or a pointer to a graphics port.

The mask that filters out the unwanted pixels is a bitmap or a pixmap. Either is passed as a pointer to a bitmap. Normally in game programming you'll just use a bitmap as the mask. With a bitmap as a mask `CopyMask` will only copy pixels from the source pixel image where the mask has black pixels.

If the mask is a pixel map of a depth greater than 1 bit, `CopyMask` will transfer the source pixel if the mask pixel is black and leave the destination pixel alone if the mask is white. Any other color in the mask will generate an output pixel that is a weighted

average between the source and destination pixels. The weighting is done using the color components of the pixels and the mask according to the formula

```
(mask - 1) * source + mask * destination
```

While the blending effects that can be produced using a color mask can be spectacular, speedy they are not. For game programming stick with 1 bit black-and-white masks.

The source and destination bounds of the pixel copy are passed as rectangle pointers. The rectangle that defines the mask bounds needs to be the same size as the source rectangle. Again as with `CopyBits`, you can scale the image as it is copied by setting the source and destination rectangles to the correct ratio. If you do scale the image, make sure you adjust the mask rectangle to match the source rectangle—or suffer the debugging consequences.

CopyDeepMask

Sometime during the development of 32-Bit QuickDraw, an Apple engineer was struck with the idea of combining `CopyBits` and `CopyMask`.

From `CopyBits` the transfer mode and region mask were grafted onto the end of the parameter list for `CopyMask`, giving the world `CopyDeepMask`.

```
void CopyDeepMask(const BitMap *srcBits,
                  const BitMap *maskBits,
                  const BitMap *dstBits,
                  const Rect *srcRect,
                  const Rect *maskRect,
                  const Rect *dstRect,
                  short mode,
                  RgnHandle maskRgn);
```

The number of ways you can muck with pixel maps using `CopyDeepMask` is a really big number, and none of them are of any use for game programming. They're great if you're writing the next

PhotoShop, but not an arcade game. For blitting stick with the simpler `CopyBits` or `CopyMask`.

Blitting Speed

In order to use either `CopyBits` or `CopyMask` as your game's animation engine, you'll need to squeeze out every drop of performance from them. By setting some parameters in one manner you can get monstrous gains in blitting speed; tweak the same parameters in a different way and you'll be able to watch each pixel draw on-screen. Understanding how these two traps work will give you the insight needed to push the blitting envelope.

Speeding Up CopyBits

The key to achieving the maximum speed with `CopyBits` is understanding `CopyBits`. You must become one with `CopyBits`. Think like a bit, be a bit. Or you could skip the Zen portion of the lesson and look at the nifty flowchart (Figure 4-4) that describes the steps `CopyBits` can pass through on the way to putting pixels on your screen.

Looking at the flowchart you can see that the path to speed is to bypass all those darn steps that the flowchart shows. Your goal in using `CopyBits` is to try to get to the "transfer" step directly, skipping all the other time-consuming steps in between.

To paraphrase Lord Kelvin, "To understand `CopyBits` you must measure `CopyBits`." In following Lord Kelvin's advice I've measured how long it takes `CopyBits` to get through its various machinations. With these timings you can see how certain optimizations can dramatically improve the blitting speed of `CopyBits`.

The timings for `CopyBits` were carried out on my Quadra 700, running System 7.1.1. The QuickDraw version is listed as 2.30. All tests were performed with the monitors set to 8 bits per pixel.

The benchmark consisted of timing `CopyBits` as it copies a chunk of pixels 256 ˘ 256. This blit is performed 100 times for each test with the results measured in ticks.

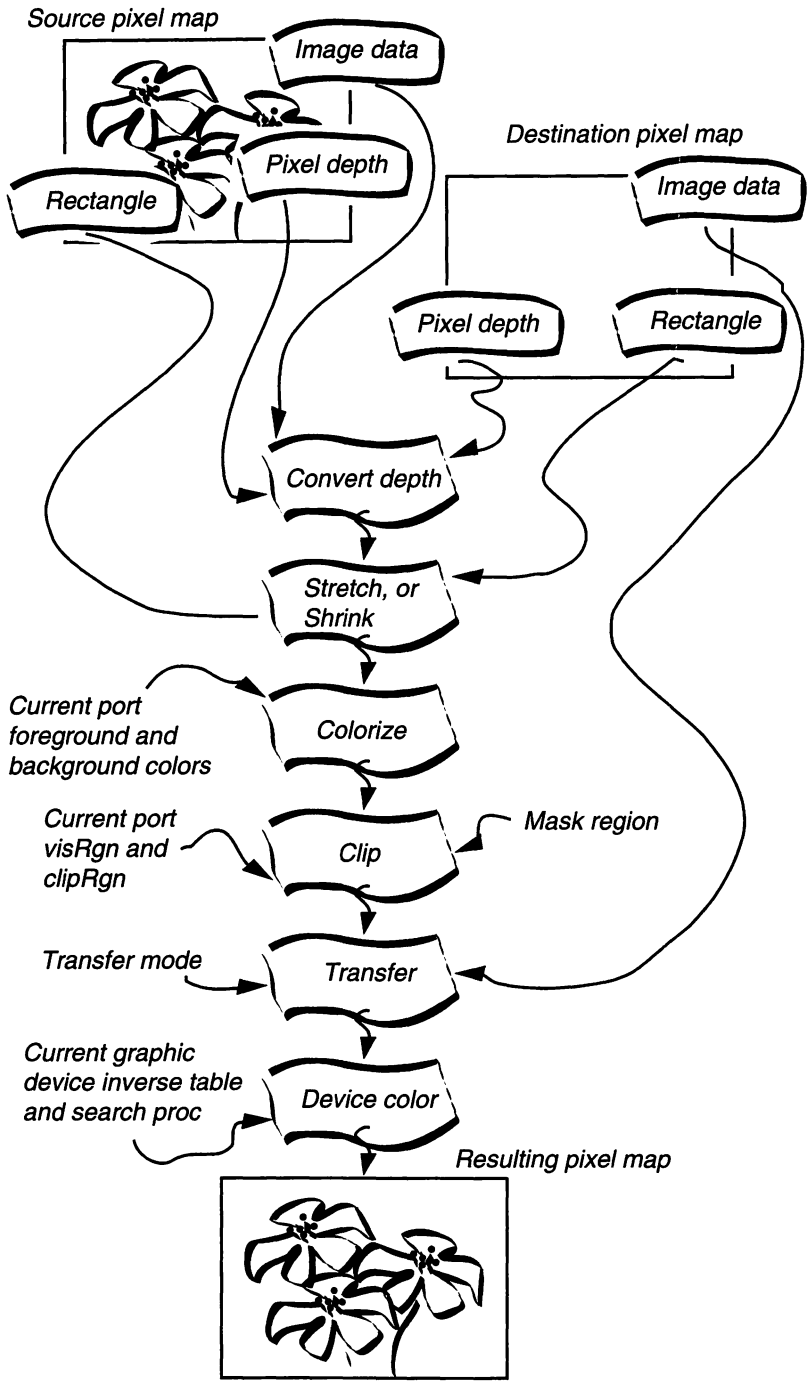


Figure 4-4. CopyBits flowchart

To normalize the results of the tests, a straight memory copy of the same number bytes as `CopyBits` will have to blit (256 ˜ 256, or 65,536 bytes) is performed using `BlockMove`. The `BlockMove` function is used to establish how fast a perfect blitter could move the pixels. On my system `BlockMove` is able to transfer 65,536 bytes 100 times in 28 ticks. Not bad for a machine that has been rode hard and put up wet too many times.

The perfect blitter can move the test chunk 100 times in 28 ticks, but how does a straight `CopyBits` fare? Using a straight `CopyBits`, my machine can copy the test pixel image 100 times in 95 ticks. The straight `CopyBits` copies from an offscreen GWorld to on-screen with no scaling, a transfer mode of srcCopy, and passing nil for the masking region. The offscreen GWorld uses the screens GDevice for its color table and has the same pixel depth as the screen, 8 bits per pixel.

Now that a baseline measurement has been established, you can see how the various optimizations to `CopyBits` impact its blitting performance. First, let's examine the trivial ways to increase `CopyBits'` performance.

Blitting Bounds

The easiest way to accelerate `CopyBits` is to reduce the number of pixels it has to haul around. In my test I reduced the offscreen and on-screen bounds by half, leaving a blitting bounds of 128 ˜ 128 pixels. With these smaller bounds a straight `CopyBits` executed 100 times in 39 ticks. Interesting that copying quarter of the pixels was done in 41 percent of the time as the baseline `CopyBits`. The results of this test can be placed in the excruciatingly obvious category. It doesn't take a rocket scientist to figure out that how many pixels `CopyBits` has to copy is proportional to how long it takes to copy the pixels. But you never know if you don't measure.

If you remember the code for the trivial blitter that was presented earlier, you might be able to predict that `CopyBits'` blitting of a bounds that is wider will be faster than its blitting of one that is taller. The theory is that `CopyBits` will spend less time figuring out where the next row starts and more time copying the pixels in a

row. To test this theory I altered my test to copy a rectangle that is four times as wide as high, 512 wide by 128 high in this case. The same number of pixels will be copied, just a fewer number of rows. The test produced a result of 93 ticks to copy the wider image 100 times. Not much of an improvement, but faster is faster.

Conclusion? You can speed up `CopyBits` by reducing the number of pixels it has to copy or by making sure those pixels are contained in as few as rows as possible.

Pixel Depth

The next obvious way to increase the speed of `CopyBits` is to move fewer bytes. Reducing the bounds reduces the number of pixels to be copied, which reduces the number of bytes. Likewise, reducing the pixel depth reduces the number of bytes to copy. A test using an offscreen at 4 bits per pixel blitting to the screen at a matching pixel depth results in a timing of 63 ticks. Halving the bytes does not result in halving the time. This is because `CopyBits` has to deal with partial bytes. Processors hate dealing with partial bytes, and they vent their aggressions by slowing `CopyBits` down. Still, reducing the pixel depth does reduce the time, although the reduction might not be directly linear.

A pixel depth of less than eight is used primarily in games that need to move screenfuls of data at once, such as scrolling platform games or flight simulators.

Clipping

Now that we've explored the obvious ways to help out `CopyBits`, it's time to look into the more subtle factors that affect `CopyBits`' performance. The first one you'll examine is the impact of clipping.

Looking at the Clipping stage in the `CopyBits` flowchart, you can see that clipping is only involved when `CopyBits` is moving the pixels to a graphics port on-screen or when a mask region was passed in as the last parameter.

In the first case the clipping is done with the visible region and the clip region of the current graphics port. This clipping only affects

CopyBits when it's moving pixels to a port on-screen. Like any other call to QuickDraw, the results are clipped to the intersection of the visible region (visRgn) and the port's clipping region (clipRgn).

The second clipping situation arises when CopyBits is passed in a mask region. A mask region given to CopyBits will mask or clip any pixels that don't lie within the masking region. If the blitter is moving pixels to a port on-screen, the final clipping region will be an intersection of the clip region, the visible region, and the mask region.

To measure the speed of CopyBits when using regions, the benchmark creates a mask region that inscribes the pixel bounds of the blit. One hundred CopyBits operations with this mask results in a timing of 111 ticks. Not bad, a paltry 15 percent increase in time with a good-sized mask. CopyBits never ceases to amaze me.

A region is an arbitrarily complex shape that impacts the speed of CopyBits directly in proportion to the region's complexity. I define a complex region as one that is not made up of strictly horizontal and vertical edges. A circular region qualifies by my definition as a complex region, but let's see how CopyBits does with a doozy of a mask region (Figure 4-5).

Using the doozy region as a mask for CopyBits gives a time of 153 ticks. Still darn respectable given the region CopyBits has to struggle with.

Conclusion? To reduce CopyBits' clipping effort, don't pass in a mask region. And if you must pass in a mask region, pass in a simple one. Controlling the clipping when it comes to the graphic port's region gets a little more complicated. Well, slightly more complicated, since when you have control of the clip region of the graphic port you can just set it to match the bounds of the port, effectively removing it from the clipping equation. That leaves the visible region of the port. *Inside Mac* warns you repeatedly not to mess with the visRgn—that would be bad. To remove the visible region as easily as the clipping region you have to understand how the visible region is changed by the Mac.

The visible region of a port defines the area of the port that is visible on-screen. When you move one window partially in front of another window, the visible region of the lower window is altered

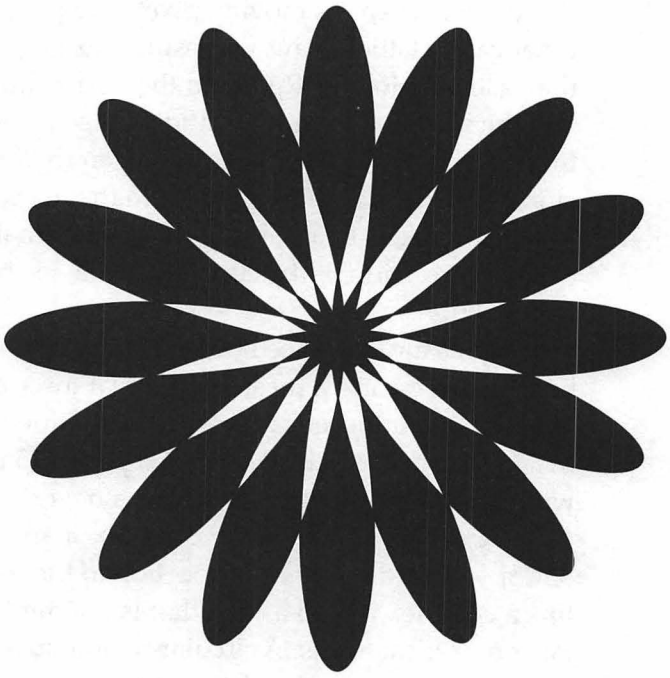


Figure 4-5. A doozy of a mask region

to remove the area where the upper window intersects the port. Knowing this, the answer is simple: make sure the window that is the destination of the `CopyBits` is the topmost window and is not overlapped by any other windows. Then the visible region will coincide with the port bounds and `CopyBits` will be free to start slamming pixels around with minimal hindrance from clipping.

Forcing a window to be topmost and not overlap any other window would be difficult to do in a normal Mac application. Your users wouldn't stand for it. Luckily, this same restriction is widely accepted as the norm for games on the Mac.

Color Mapping

The last stage of `CopyBits` before the pixels appear on-screen is to map the colors from the source color table to the destination color

table. This is a very costly operation. Using `CopyBits` to copy from an offscreen with a different color table than the one the screen uses results in a timing of 228 ticks. From 95 ticks to 228 is a big decline in blitting performance and one to be avoided.

If the color tables of the source and destination pixel maps don't match, `CopyBits` has to color map each pixel from the source color environment to the destination. To color map one pixel `CopyBits` first converts the index value of the pixel to an RGBColor structure using a process similar to the one used by `Index2Color`. After getting the source pixel as an RGBColor structure, `CopyBits` then finds the index of the destination color table that belongs to the current graphics device, coming closest to the matching source color. This RGB to index is done through the Toolbox function `Color2Index`. The `Index2Color`, `Color2Index` process is done for each and every pixel `CopyBits` is asked to copy. Sounds slow. It is. Avoid it.

The secret to avoiding color mapping is not to have any colors in your offscreen color table that are different than the colors in the color table that the current GDevice owns. Plus, you need to tell `CopyBits` that the two color tables are the same and that color mapping would be a waste of your precious time. `CopyBits` determines that two color tables are different by checking the ctSeed field of the color tables. If the ctSeeds of both tables match, `CopyBits` assumes that the colors in them are the same, skips the color mapping, and jumps right to the pixel-blasting stage. If the ctSeeds don't match, `CopyBits` scans each color in the source color table, checking to see if the color matches the one in the destination color table. If each color has a matching color in the destination table, `CopyBits` is again free to start transferring pixels in overdrive without worrying about mismatching the colors. If both tests fail, `CopyBits` starts the laborious process of color mapping each pixel as it is copied.

Color mapping within `CopyBits` is only a problem with indirect pixel maps. Pixels in a direct color pixel map are transferred by `CopyBits` without interference from any color mapping. Too bad direct color maps are so darn big and slow. They would save you the worry that color mapping will slow down your game. Conclusion? Make sure the colors in your offscreens match the colors used by the current GDevice. And don't forget to tell `CopyBits` about

all the trouble you went to by setting the ctSeeds of all the color tables to the same value. The easiest way to assure that all the color tables are set correctly is to use the GWorld routines and let them worry about it.

Black and White

A separate stage from color mapping is `CopyBits`' colorizing step. Right before the clipping stage, `CopyBits` checks the foreground and background color of the current graphics port. If the foreground and background are not black and white, `CopyBits` figures you want to colorize the pixels as they are copied and that you are not in a hurry about it. In the `CopyBits` test with the foreground color to red and background set to blue, the 100 times loop takes 241 ticks.

Colorizing usually happens in a game as a bug. You set the foreground or background color in order to draw in to the game window (drawing the current score for example) and forget to restore the foreground and background to black and white when you're done. The next thing you know your next call to `CopyBits` screeches into slow-mo.

No Scaling

It's obvious that scaling the pixel image through `CopyBits` is a sure path to glacial game performance. But it's that path that's often taken. The situation I'm referring to is where you have an "off by one" error in your code that calculates the source or destination rectangles. This error is not always visible but has a dramatic impact on performance. So if your game is running sluggishly and everything else seems sane, verify the blitting bounds. You won't be the first person to be snagged by this problem.

Transfer Modes

The table lists the timings for copying the standard block of pixels with each of the transfer modes that `CopyBits` supports. If you

ever have a thought about using one of the transfer modes within your core animation engine, Table 4-1 will quickly chase that dangerous thought out of your head.

Table 4-1. Transfer Modes Timings

srcCopy	95	notSrcOr	383	subOver	926
srcOr	111	notSrcBic	382	subPin	1087
srcBic	202	notSrcXor	210	adMax	1064
srcXor	217	addOver	928	adMin	1005
notSrcCopy	681	addPin	1026	blend	1747

The blend transfer mode takes 1,747 ticks, close to five seconds. Yikes, that would be one slow blit. You could hand-draw each pixel on the screen faster than `CopyBits` performing a blend.

Conclusion? Easy. Transfer modes are evil. Don't use them in anything that you want to happen in your lifetime.

Memory Alignment

When working with a processor such as the Motorola 68020 and above, you need to keep in mind how data moves in and out of the processor. The processor wants to munch on the data in four-byte gulps, with these gulps starting on long word boundaries. If the data being moved is smaller than four bytes, the processor has to shift out of its natural stride to handle the smaller parts of data. This shifting in and out of high gear results in a speed reduction.

By forcing your pixel maps to lie on long word boundaries, you'll get another increase in `CopyBits`' speed. The straight `CopyBits` test altered for long word alignment gives a timing of 58 ticks for the 100-count loop. That's a mammoth increase in speed for being a little more careful about how the rowBytes is calculated.

You don't even have to be that careful. Pass `NewGWorld` with a pixel depth of zero and `NewGWorld` will hand you back a GWorld with long word alignment.

Conclusion? Make sure your pixel data is long word aligned. You'll have to look hard to find an optimization that almost doubles `CopyBits`' speed that is as cheap as long word alignment.

Avoid the Screen

If you follow all the performance suggestions in this section, you'll reduce the stages `CopyBits` has to pass through down to two, clipping and transferring of the pixels. You can't get rid of the transfer stage—well, you could, but then why use `CopyBits` when a NOP will work? That leaves the clipping stage. As long as you're moving the pixels to a position on-screen, you're stuck with that stage and the best you can do is reduce the amount of clipping to be done.

Now, if you aren't moving the pixels to a position on-screen, you're talking a whole new ball game. Without clipping to worry about, `CopyBits` can leap right to the transfer stage. How new is the ball game for a pixel map to pixel map transfer? I altered the standard test to copy from one offscreen to another offscreen. Transferring the pixels from the two maps 100 times with `CopyBits` gave a timing of 39 ticks.

Conclusion? Removing clipping gives back a huge increase, right? Not quite. This statistic can be misleading. The speed increase is due more to where the pixel images exist in memory than to avoiding the clipping stage. The next section goes into more detail on the memory differences. For now, chalk up a slight increase in performance to the removal of the clipping stage and read on to find out where the rest of the time went.

Hardware Speedup

Want an easy way to speed up `CopyBits` with no work on your part? Convince your players to buy faster Macs. The odds of that occurring are slim unless you're writing games for Microsoft. Another way to faster hardware is to find some that is faster that already exists in your players' machines.

To prove this nebulous point I altered my tests to use the built-in video instead of my video display card (see Table 4-2). Results?

Table 4-2. NuBus Card versus Built-in Video

CopyBits Option	*Radius DirectColor/GX*	*Macintosh Quadra Built-in Video*
Normal CopyBits	95	46
Aligned CopyBits	58	36
Half-Size CopyBits	39	14
Wider CopyBits	94	46
Region CopyBits	217	67
srcOr	111	114
srcBic	202	107
srcXor	217	112
notSrcCopy	681	595
notSrcOr	383	152
notSrcBic	382	155
notSrcXor	210	112
addOver	928	806
addPin	1026	900
subOver	926	804
subPin	1087	969
adMax	1064	941
adMin	1005	888
blend	1747	1628
Colorize	241	224
Different Color Table	228	211
GWorld To GWorld	39	39

Dramatic: the straight `CopyBits` test that took 95 ticks with the display card timed out to a scant 46 ticks with the built-in video. The long word aligned test dropped from 58 to 36 ticks. Almost doubling the speed of `CopyBits` without having to upgrade the hardware is a pleasant surprise.

When CopyBits is transferring pixels to video RAM on my display card, it has to move that data across NuBus. The NuBus is clocked at 16 Mhz and has a top transfer rate of 10 megabytes per second. When the first Mac II was designed the NuBus specification was an ideal match for the 16 Mhz 68020. The trouble began when Motorola started producing faster processors. What once was a nicely balanced design began to suffer from I/O bottlenecks.

My Quadra 700 has a 25 MHz '040 that is still tied to that same old NuBus. So whenever CopyBits has to move pixels over to my NuBus video card, the processor is kept waiting until NuBus has transferred the data. The built-in video of my Quadra 700 is not located on a NuBus slot and does not suffer for its sins. When CopyBits directs its attention to the built-in video, it can do so at the full bandwidth the processor is capable of.

Your players might not have built-in video, but they might have invested in a QuickDraw acceleration card. These NuBus display cards are intended to off-load the processor from having to perform all the graphics grunt work and shift that work load to dedicated processors on the display card. The work load shifting has three benefits: first, the main processor has more time to do other nongraphic operations, like play sounds. Second, the dedicated silicon is usually faster at graphic operations than the general purpose processor it replaces. And last, most accelerators cache the pixel data on the display card in order to avoid the NuBus bottleneck.

Conclusion? You need to test CopyBits against whatever blitter you might have concocted. On some of your players' systems your custom blitter might easily be outperformed by the built-in hardware. Plus, if your users paid for the hardware, you should be nice enough to use it.

Care and Feeding of CopyBits

In an attempt to sum up, let's go over the care and feeding of CopyBits to insure blissful blitting.

- Minimize the number of pixels copied
- Minimize the pixel depth
- Minimize the amount of clipping
- Match color tables and ctSeeds
- Avoid scaling
- Avoid transfer modes—they are evil
- Take advantage of long word alignment
- Time `CopyBits` to maximize hardware
- Do not feed after midnight
- Keep away from water

Speeding Up CopyMask

Increasing the performance of `CopyMask` should be relatively easy given that `CopyMask` is a close cousin of `CopyBits`. All the tips and tricks used for `CopyBits` can be used with `CopyMask`.

The flowchart of `CopyMask` (Figure 4-6) adds a new step: "Combine with mask." In this step the destination pixels are determined from the intersection of the mask, the destination pixels, and the output of the scaling stage. The output from the mask stage is passed along to the clipping stage, and processing continues exactly as in `CopyBits`.

The same test that was used for testing `CopyBits`' masking capabilities—masking out a circle that inscribes the bounds—is used to test `CopyMask`. Instead of a region, a 1-bit-deep offscreen with a black circle painted inside is used as the mask passed to `CopyMask`. As with `CopyBits`, the `CopyMask` test is run a hundred times with a resultant timing of 369 ticks.

When I first ran this test I was slightly taken aback by the results. The time of 369 ticks with `CopyMask` seemed strange when compared to the test with `CopyBits` that resulted in a time of 111 ticks. I couldn't believe that `CopyBits` with a region could be three

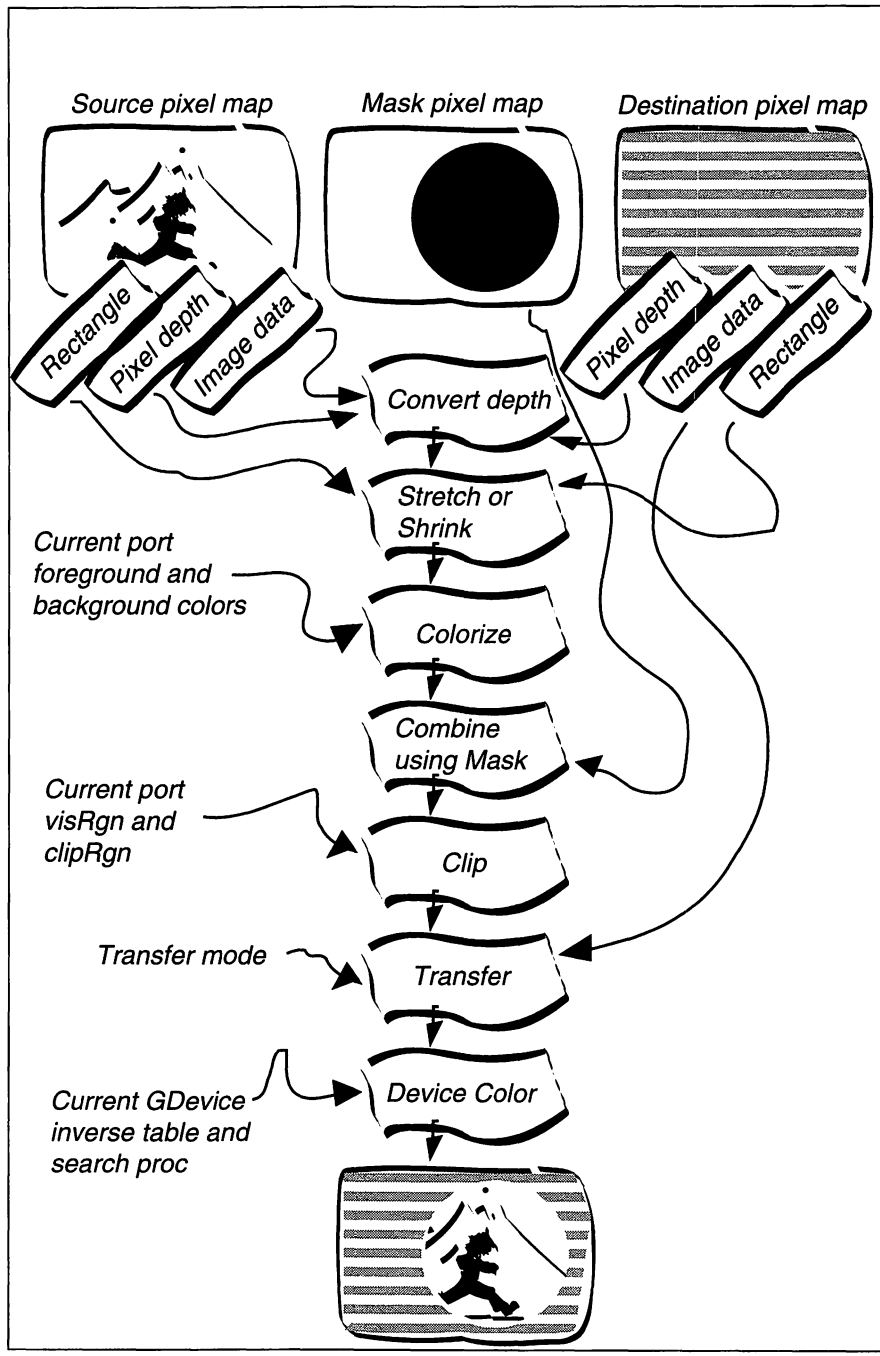


Figure 4-6. CopyMask flowchart

times faster than CopyMask. Luckily, I asked other programmers about my apparent mystery. I was looked at as if I had asked if they had noticed that the world was round. It seems Apple had announced this stunning finding that CopyBits with a mask region was significantly faster than CopyMask a while back, when I wasn't paying attention. This tidbit of useful information was contained in a sample snippet called "CopyBits vs. CopyMask." Guess the outcome.

Even with Apple engineering-generated proof, I still resisted the evidence. My theory was that CopyMask was spending most of its time trying to extract the bits that make up the mask. When using masks it's common to expand the mask so that each mask value takes up one byte instead of one bit. The expansion allows the blitter's loops to move along the mask at the same pace at which it moves through the source pixels. At least that was my theory. To test my theory I altered my test to use an 8-bit-deep mask in conjunction with CopyMask. Results? For the simple circle mask, the 8-bit-deep mask produced a time of 3,500 ticks. 3,500 ticks, that's close to a minute. I've booted, played, and lost games in less than a minute.

As a last, desperate act to prove my theory, I altered the test to use the infamous "doozy" picture as a mask for CopyMask. The doozy mask generated a time of 3,485 ticks. A 15-tick speedup doesn't account for much when the blit takes over 58 seconds.

Conclusion? Let's see . . . CopyBits with a circle region takes 111 ticks to mask, while CopyMask clicks off in 369 ticks, or you can blow eight times as much memory with a deeper mask and get a time of 3500 ticks. My conclusion is to get out your copy of *Inside Mac*. Find the section for CopyMask. Grab a big red marker and write "Skip it. Use CopyBits."

Unmasking Regions

I accepted that CopyBits is the superior A-Trap. What I wanted to know now was how. What's in a region that's better than a classic mask bitmap? Grabbing my freshly annotated copy of *Inside Mac*, I looked up the definition of a region. Once again I found the Book of Revelations more direct than this entry in this book.

```
typedef struct Region {
      short rgnSize;
      Rect  rgnBBox;
} Region, * RgnPtr, **RgnHandle;
```

Inside Mac gives a description on how useful regions are and says that for your own safety you don't need to know how they work internally. I was bored that weekend and decided to find out what exactly a region is composed of.

My starting point was finding and looking at a few regions under the debugger. A region made from a square 32 pixels on a side produces a region with no data (see Figure 4-7).

Figure 4-7. Simple region

```
Region Raw Dump: 000A 0000 0000 001F 001F
Region Size: 10
Region Bounding Box: 0,0,31,31
Region Data : empty
```

Rectangular regions are indicated by a size of 10 bytes. The region's bounding box then defines the actual region. Since the region is fully described from the size and the bounding box, its data is empty.

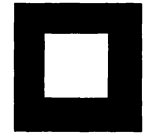


Figure 4-8. Square wheel region

The next region I looked at was the same rectangle with a hole in the middle (Figure 4-8). This produced a region of some interest, as the data dump shows.

```
Size:           44
Bounding Box:   0,0,31,31
Data:           0000 0000 001F 7FFF 000A 000A 0015 7FFF
                0015 000A 0015 7FFF 001F 0000 001F 7FFF
                7FFF
```

I figured I needed to look at a region that had some curvature to it. So I created a wheel the same size as the other regions (Figure 4-9). Its data dump provided the most insight into the region data internal structure.

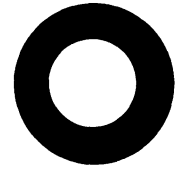


Figure 4-9. Ordinary wheel region

```
Size:           324
Bounding Box:   0,0,31,31
Data:           0000 000C 0013 7FFF 0001 0009 000C 0013
                0016 7FFF 0002 0007 0009 0016 0018 7FFF
                0003 0006 0007 0018 0019 7FFF 0004 0005
                0006 0019 001A 7FFF 0005 0004 0005 001A
                001B 7FFF 0006 0003 0004 001B 001C 7FFF
                0007 0002 0003 001C 001D 7FFF 0009 0001
                0002 001D 001E 7FFF 000A 000D 0012 7FFF
                000B 000C 000D 0012 0013 7FFF 000C 0000
                0001 000B 000C 0013 0014 001E 001F 7FFF
                000D 000A 000B 0014 0015 7FFF 0012 000A
                000B 0014 0015 7FFF 0013 0000 0001 000B
                000C 0013 0014 001E 001F 7FFF 0014 000C
                000D 0012 0013 7FFF 0015 000D 0012 7FFF
                0016 0001 0002 001D 001E 7FFF 0018 0002
```

```
0003 001C 001D 7FFF 0019 0003 0004 001B
001C 7FFF 001A 0004 0005 001A 001B 7FFF
001B 0005 0006 0019 001A 7FFF 001C 0006
0007 0018 0019 7FFF 001D 0007 0009 0016
0018 7FFF 001E 0009 000C 0013 0016 7FFF
001F 000C 0013 7FFF 7FFF
```

After spending some time contemplating these data dumps. I gave up unraveling the innermost mysteries of regions by myself and asked some friends if they knew the internal layout of a region. And of course they all did. Every last one of them. I hate having smart friends.

Regions turn out to be a form of compressed bitmaps. The compressing is done through run length encoding the binary data. For those without the benefit of a classical education, run length encoding is a compression method that looks at a length of data and replaces runs of like data with a count of the items in the run. With data that has a high level of redundancy, like bitmaps, you can replace a lot of bits with just a few bytes. Regions then go beyond simple run length compression by taking advantage of the fact that in bitmaps the next row of pixels is very likely to be similar to, if not an exact copy of, the previous row. With run length compression applied to both dimensions, regions squeeze every bit of redundancy out of a bitmap.

Let's walk through the first image with a hole and see how the region falls out (see Figure 4-10).

```
Data:     0000 0000 001F 7FFF 000A 000A 0015 7FFF
          0015 000A 0015 7FFF 001F 0000 001F 7FFF
          7FFF
```

Quick overview first. Each chunk of a region starts with the vertical coordinate of the scan line being compressed. Following that entry are horizontal coordinates that indicate where a pixel run starts and stops. The value 0x7FFF is used as a sentinel to signify that there is no longer any data in this scan line. Using the above road map, the first scan line is

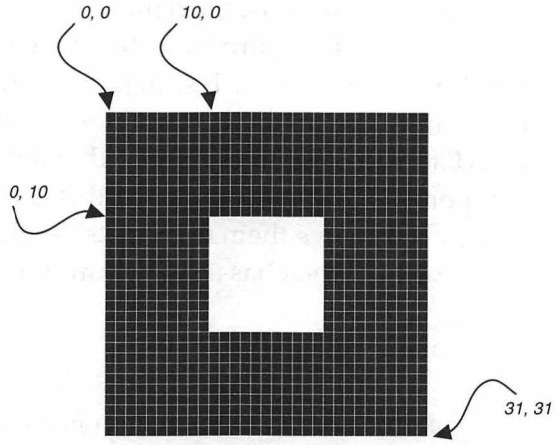


Figure 4-10. First image with a hole (detail)

```
0000 0000 001F 7FFF
```

and decodes as starting at coordinate 0 with only one run of pixels starting at pixel 0 and ending at pixel 31.

```
000A 000A 0015 7FFF
```

The second entry of the region starts at scan line 10. Which means that all scan lines between 0 and 10 are duplicates of the first scan line and don't need to be listed in the region. The second entry in this chunk starts at position 10, which is the beginning of the hole. Now, obviously this "10" isn't where a run of pixels starts. So what's going on? The snag here is that any run of pixels inherits the run values of the previous scan lines, in this case the values 0 and 31. With these inherited values the pixel runs for scan line "10" turn out to be 0 to 10 and 21 to 31. The next entry in the region exposes the last technique that regions use.

```
0015 000A 0015 7FFF
```

On scan line "15" the hole in the bitmap ends. So shouldn't the encoding for this scan line be the same as the first scan line, pixel 0

to pixel 31? Trouble is, the data we have is the same as scan line "10," not "0." This introduces the last encoding method that regions use. Remember that each scan line inherits the horizontal run indicators from the previous scan lines, so this scan line inherits the runs 0 to 10 and 21 to 31. Notice that the scan line declares two components that it also inherits; this is region-ese for any two like values, so remove them from consideration. According to this new rule, line "15" then has a single run from 0 to pixel 31.

```
001F 0000 001F 7FFF 7FFF
```

The final section of the region is now a breeze. Scan line "31" writes out the last line of the region and then ends the region with the standard region epilogue: 7FFF to end the line and 7FFF to end the region. Every region ends with 7FFF 7FFF.

Now back to the original reason for this departure into the inner workings of regions. Why is CopyBits with a mask region faster than CopyMask? The answer lies in the run lengths. Each scan line of the region has the information embedded within it to describe to CopyBits just which pixels to copy over and which pixels to skip. And more important, the pixels that need to be copied are described in whole chunks. Processors love to move chunks of pixels. While CopyBits is blasting pixels like there's no tomorrow, CopyMask has to process every single pixel, performing those awful masking operations on each of them, including the pixels that won't end up being transferred. What a waste of time. Add all these obstacles up and CopyMask just runs into too many speed bumps to keep up with CopyBits. You'll take this keen insight into account when you start writing you own replacement for CopyBits. That sounds a little ominous, doesn't it? Without any further foreshadowing let's head right for the heart of game programming. Sprites.

5

Sprite Blitting

Sprite?

A long time ago, in a computer far, far away, some nameless yet talented engineers created some nifty hardware blessed with the ability to move small chunks of pixels around on a video screen very quickly. These small chunks of pixels could be moved around the screen without disrupting the video image underneath. Even better, these pixel chunks could be moved just by setting a few registers. Change a few other registers and the hardware would overlay a different chunk of pixels on the screen. As a final bonus the hardware could even detect when two of these pixel chunks overlapped.

These engineers couldn't go around talking about "chunk of pixels this" and "chunk of pixels that." They'd have seemed silly, well at least sillier. So according to game programming folklore, these dutiful engineers looked at the on-screen splotches that could be barely distinguished from one another and dubbed them *sprites*.

Since then, any animation system, whether implemented in software or hardware, that gives a programmer the facility to move multiple, overlapping chunks of pixels on the screen without destroying the background screen image has been referred to as sprite animation. Except for Atari, which named its sprite systems *player-missiles*. Which while more accurate doesn't have any pizzazz. Along with moving your pixels around, a sprite system will throw in the ability to have multiple frames of animation for each sprite, notification when two sprites touch, and with multiple sprites, a method of setting the order in which the sprites are drawn.

In this chapter you'll be introduced to the techniques used to build a sprite system, or engine. You can think of this chapter as the salad bar of sprite techniques, from which in a later chapter you'll reach under the sneeze guard to pick out just the perfect, juicy parts you'll need.

Anatomy of a Sprite

Rip into any sprite, on any system, and you'll find pretty much the same innards.

- Frame or series of frames of animation that represent the sprite
- Mask for each frame in the sprite
- Bounding rectangle for each frame
- On-screen location for the sprite
- Speed and direction for sprite movement
- Drawing priority

Let's use sprite "Tommy" here as an illustrative example (Figure 5-1). There are four separate frames of animation for Tommy. As you cycle through these frames Tommy appears to run.

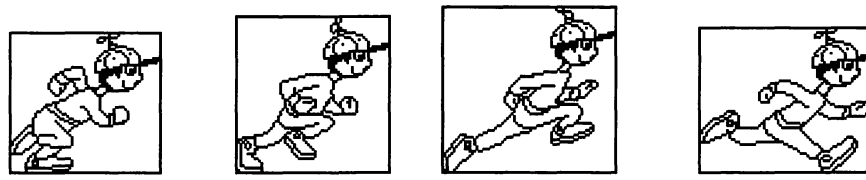


Figure 5-1. Tommy

Tommy, as a sprite, has two needs that are tied to time: how many pixels per second should he move, and at what rate will he cycle through his frames of animations. Without any velocity, Tommy will run in place. Giving Tommy a horizontal velocity of 6 pixels per second (vertical velocity for Tommy is set to zero; he doesn't have much vertical lift), he will cycle through all of his frames of animation three times a second, giving an effective frame rate of 12 frames per second.

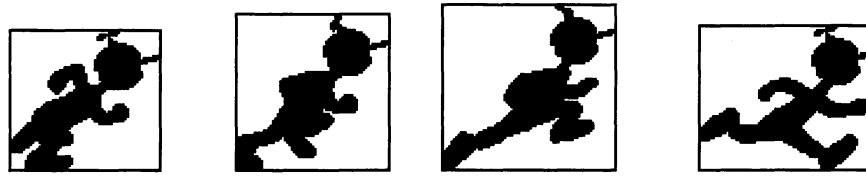


Figure 5-2. Tommy's masks

For each frame of animation Tommy needs a matching mask (Figure 5-2). For sprites whose outline does not change between frames, such as a revolving moon, a single mask will do for all the frames of animation. The masks don't have to be bitmaps; they could as easily be stored as regions.

Surrounding each of Tommy's frames is a bounding box. The bounding rectangle is the minimum bounds that encompass each pixel in the frame. This frame is used by the blitter and other parts

of the sprite engine. Notice that each of Tommy's frames does not have to fit within the same size bounding box.

Tommy will have to be positioned somewhere on-screen. His position could be stored as a point. But what would the point refer to? It could refer to any point in the sprite, or it could even refer to a point outside of the sprite, but that would be a pain. Some sprite systems use the upper left corner for the sprite's position. Others use the lower left corner. The style is determined by the type of game being programmed. A game where the sprites have to walk and jump on solid platforms à la Mario Brothers usually will pick the bottom left. For Tommy we would choose also to use the bottom left. Why? If we picked the top left, Tommy's feet would not be on the ground for any frame other than the first one. Which would be O.K. if Tommy were running on the moon but would seem out of whack for the run-of-the-mill earthbound sprite.

The last essential element a sprite must have is priority. In sprite systems priority refers to the drawing order of the sprites. A sprite with a higher priority will be drawn last and on top of all the other sprites with lower priorities. The sprite with the lowest priority will be the one on the bottom. The priority can be changed for each sprite, but all of the sprite's frames share the sprite's priority.

Sprite priority becomes important when you need the illusion of depth in your game. If you want Tommy to walk behind a fence, a car, or a 40-foot T-Rex, you need to set Tommy's priority to a lower value than any of the sprites he needs to walk behind. Otherwise Tommy will be stepping on some T-Rex toes. And when the T-Rex is chomping on Tommy's beanie-topped noggin, make sure you set T-Rex's chompers' priorities to one higher than Tommy's.

Hardware Sprites

Before diving into the software implementation of sprites let's take a quick tour of a hardware-based sprite system. Warning! After reading this section, every time you have to write some sprite code you'll end up screaming in the direction of Cupertino: "Hey Apple, if you're serious about supporting the home market how about

some hardware support. Stop wasting your time with things like Geoports and Newtons and build sprite hardware into the next Mac. Jeez C-64s had 'em way back . . ." Slow fadeout into a Commodore nostalgia. Well, at least that's what happens with me. Hopefully you'll have more self-control.

A hardware sprite system relieves the main processor of two tasks: blitting the sprite image onto the screen and detecting when two sprites collide. All the other features of a sprite engine have to be provided by the programmer or operating system. But this is a fair trade, as these two tasks are where software sprite engines burn most of their cycles. With the main processor now freed from the burden of blitting, it can go off and do more important stuff. Like making annoying sounds. Though usually by tossing in a sprite processor the hardware designer can now use a slower, cheaper microprocessor for the heart of the system.

With a hardware blitter the programmer only has to provide a pointer to where the pixel image for the sprite is stored, punch in the on-screen location for the sprite into a couple of registers, and stand back. The single frame of the sprite is now on-screen. Change the location registers and on the next refresh cycle of the screen the sprite will have moved. Place a pointer to a different sprite image and the sprite changes. Do both at the same time and you have a moving, animating sprite.

So what technological marvel does the hardware perform that software cannot? Unlike most hardware versus software implementations, the hardware is not just a faster version. A hardware blitter actually does blitting a better way. Remember back a hundred pages or so, there was a discussion of how a raster display works by converting the frame buffer memory into analog signals that produce your picture. A hardware blitter works downstream from this frame buffer scanning. As the video memory is being processed, the blitter detects when the scanning process is on a scan line that a sprite intersects. When this happens the blitter switches from scanning pixels out of the frame buffer to scanning them from the area addressed by the pointer you installed in the sprite's pixel image register. When the current scan line leaves the bounds of the current sprite, it falls back to scanning the frame

buffer. This scanning process is performed for each scan line that a sprite intersects and repeated for every refresh cycle of the monitor. The truly neat part, at least to me, about this process is that the underlying image in the frame buffer remains untouched by the sprites.

Hardware blitters, just like other blitters, need to provide for masking the unwanted pixels of the sprite. Older hardware blitters performed the masking by treating one of the colors in the pixel image as transparent. When the blitter is scan-converting the sprite, it checks each pixel value and when it crosses a pixel that has the same value as the designated transparent color, instead of converting that pixel it reaches back into the frame buffer and writes the pixel found there to the screen. More modern blitters have a separate pointer to a pixel image that is used as the mask for the pixel image. The pixel image used for the sprite's mask doesn't just encode which pixels are displayed or not. Each value in the mask maps to a transparency level for each of the sprite's pixels. If the mask is stored with a byte for each pixel, a value of zero would give you a fully transparent pixel. Where a mask value of 255 would make the sprite's pixel fully opaque, obscuring the background, any mask value in between will result in a semitransparent pixel. Just the thing if you need to write a haunted house game.

The other function a hardware sprite system usually provides is collision detection. While the hardware is scan-converting the sprites to the screen, it looks at each sprite and determines if the sprite intersects any of the other sprites. When a sprite collides with another sprite the hardware notifies the programmer and passes along which sprites are doing the colliding, usually through an interrupt. A collision is only detected when the two masks for the sprites intersect where the mask values are nontransparent. This allows the sprite to have holes that other pixels can travel through without triggering a collision. The hardware will usually provide another signal when the sprites are finished colliding. What happens when two sprites collide is strictly up to the programmer. The hardware reports only the facts of the collision—the cause, not the effect. That's up to the programmer.

We've reached the end of our tour into hardware blitters, and you can now see why it would be impossible to perform sprite manipulations using the same techniques. So unless System 8 gives us the ability to reprogram the Mac's video scanning, we're stuck with a severe case of hardware envy.

Who Was That Masked Sprite?

Now that you know the internals of a sprite and the inner workings of a hardware system that you'll never have access to, it's time to move on to the real works: making a software sprite engine for the Mac. The first part you'll need for your engine is a masked blitter. In order to pick just the right blitter you'll need to know your various options.

Brain Dead Again

First off in our masked blitter rundown is a variation of the "brain dead blitter" that was presented earlier. The original blitter just copied over the pixels. In this version of the blitter a pixel value is passed in that represents the color index for the pixels we want to mask. This could be hard-coded to a preordained color index, but for now we'll just pass in the transparent value.

```
void  BrainDeadMaskedBlit( PixMapPtr  srcMap,
                           PixMapPtr  dstMap,
                           Rect *     srcR,
                           Rect *     dstR,
                           char       transparentValue)
{
      char *      srcAddr;
      char *      dstAddr;
      short       srcRowbytes;
      short       dstRowbytes;
      short       row, pixel;

      // Extract the start of the pixels out of the pixmap
```

```
        srcAddr = srcMap->baseAddr
        dstAddr = dstMap->baseAddr

        // Extract the rowBytes out of the pixmap
        srcRowbytes = srcMap->rowBytes & 0x3FFF;
        dstRowbytes = dstMap->rowBytes & 0x3FFF;

        for(row = srcR->top; row <= srcR->bottom; row++)
        {
              char *      srcPtr;
              char *      dstPtr;

              // Calculate the start of each row
              srcPtr = srcAddr +
                          (row * srcRowbytes + srcR->left);

              dstPtr = dstAddr +
                          (row * dstRowbytes + dstR->left);

              for(  pixel = srcR->left;
                    pixel <= srcR->right;
                    pixel++, dstPtr++, srcPtr++)
              {
                    if(*dstPtr != transparentValue)
                          *dstPtr = *srcPtr;
              }
        }
}
```

Down in the innermost loop instead of blindly copying the pixels, the code now checks to see if the sprite's pixel matches the transparent color. If so, it's skipped and the code goes on to the next pixel. Nothing complicated here. Brute force coding in its purest form. Advantages of this blitter? Low on memory and . . . that's it. Nothing else but low on memory. By encoding the mask within the pixel image you can get a big savings in the memory area. The penalty is the classic one: speed. `CopyBits` with a region will handily wipe the floor with this blitter. Even `CopyMask` can easily outrun this blitter. So why would you use this blitter? I have no idea. Under duress, I guess.

CopyBits Redux

Next up on the blitter hit parade is the classic `CopyBits`, including a region for the mask. You already have intimate details and timings for every aspect of `CopyBits`, so let's get right to the missing detail of `CopyBits`—getting the sprite's pixel mask into a region handle.

Regions are usually built programmatically, between calls of `OpenRgn` and `CloseRgn`, like so.

```
NewRgn()
OpenRgn()
FillRect()
InsetRect()
FillRect()
CloseRect()
```

This works great if you need a region that can be built strictly out of simple geometric shapes. Try building a mask for a sprite like Tommy out of QuickDraw calls and you'll quickly create more lines of code between `OpenRgn` and `CloseRgn` than are used in your entire game. Yikes. Can't have that happening.

Reach down into the Mac Toolbox; lying right next to NewOldCall is `BitmapToRegion`. Don't you love these self-documenting Toolbox names like `BitmapToRegion`, `DisposeHandle`, and `Munger`. For an example of this nifty routine let's get a region out of the mask contained in 'cicn.' But first let's look at `BitMapToRegion` in a little detail.

```
OSErr BitMapToRegion(RegionHandle region, BitMap * bMap);
```

The region handle passed to `BitMapToRegion` must have already been allocated with a call to `NewRgn` or any other function that will give you back a region. Failure to do this will rip apart your heap or another innocent application's heap. Anytime you're dealing with regions and you start experiencing catastrophic failures, start looking for where you forgot to allocate a region.

The BitMap pointer can point to either a real bitmap or a pixel map. If you use a pixel map make sure its depth is set to one. Otherwise you'll get an error "pixmapTooDeepErr" or –148 as your debugger will show. If you pass in a bitmap that results in a region too complex to encode in 32K, you'll get a "rgnTooBigErr" or –500. Of course, if you're trying to create a region that large your sprites are probably larger than the screen, and masking is just a waste of time.

Before using `BitMapToRegion` you need to make sure that the routine exists within the Mac you're currently trying to run your game on. This routine came out with 32-Bit QuickDraw, so any test for that version of QuickDraw or better will insure that the code exists somewhere in the Mac. If you're a real stickler for operating on any system, you can always get the `BitMapToRegion` object file from Apple that you can link into your own game. Apple used to charge a $50 licensing fee for this bit of code. That is no longer the case. I can't believe they tried to squeeze a few more drops of cash out of programmers for this snippet of code. Back to the example.

```
BitMap              bitmapMask;
RgnHandle           rgnMask;
CIconHandle         colorIcon;

colorIcon = GetCIcon(kSampleIconID);
if(colorIcon)
{
      HLock(colorIcon);

      bitmapMask = (*colorIcon)->iconMask;

      // Re-point the bitmap's image Ptr to the mask data
      // in the cicn
      bitmapMask.baseAddr = (*colorIcon)->iconMaskData;

      rgnMask = NewRgn();
      if(rgnMask)
      {
            if( !BitMapToRegion(rgnMask, &bitmapMask))
            {
                  // Got the region. Do whatever you
                  // want with the region.
            }
```

```
            DisposeRgn(rgnMask);
        }

        HUnlock(colorIcon);
}
DisposeCIcon (colorIcon);
```

The snippet first gets a color icon (from where isn't relevant), and after locking down the handle wraps a bitmap structure around the icon's mask. The only thing left is a quick call to `NewRgn`. Finish with a stunning dispatch to `BitMapToRegion`. After disposing of the region and the color icon we're done. If we can nail the dismount I'm sure we can rack up a 9.9 from the coding judges.

> A quick debugging tip when working with cicn. Don't use `DisposeHandle` or `ReleaseResource` on CIconHandles. While they'll appear to work (with the proper casting to pacify the compiler), you'll have created a memory leak. `DisposeCIcon` is one of those few GetWhateverResource routines that do not return a handle to the resource data, but instead they load in the resource as a template. From this template the actual handle is created and returned to you. In the case of `GetCIcon`, it creates the handle it returns to you and a few other handles as part of the pixel map that makes up the color icon. If you call `DisposeHandle` instead of `DisposeCIcon`, you'll only be throwing out the root handle, leaving the handle that holds the icon's pixels stranded in the heap.
>
> The other gotcha is assuming that this is like any other GetResource call and that if you call it more than once for the same ID you'll always get back the same handle to the data. Each time `GetCIcon` is called it creates a fresh color icon in your heap.
>
> Failure to call `DisposeCIcon` in either if these cases will quickly deplete all of the good stuff from your heap, and next thing you know you'll be staring at a –108 error.

Now that you can rip a mask region out of just about anything that has two dimensions, your CopyBits arsenal is complete. But be wary. You'll see a lot of fancy blitters in the future, but don't fall for the high-octane, nitro-burning blitter of your dreams without first

considering whether the Volvo of blitters will do. It might not be fancy, but it's safe, reliable, and best of all, paid for. Done. No more lectures on how you should try to use CopyBits. Let's go take a look at the nasty boys of blitting.

Rolling Your Own

The trick to making a blitter that is faster than `CopyBits` is specialization. `CopyBits` has to be general in its usage. You don't. You can hard-code anything you need to gain that required extra amount of juice. Remember that as a mantra: "Hard-code for speed." While constantly repeating that under your breath, read the list of hard-coded assumptions that our blitters will be coded around.

- 8-bit indexed color
- Matching color tables
- Source and destination are the same-sized areas
- 8-bit expanded mask
- Motorola 68020 processor or better
- Assembly language is good for the soul

With these preconditions it's not much of a task to have your game blasting pixels faster than `CopyBits`.

Whenever you need to optimize code for speed that you need to work, recall the optimization credo: "Get it working. Then get it to work fast." Blazingly fast code that doesn't work isn't much better than no code at all. The other programming cliché that needs to be repeated here, if for no other reason than to reinforce it as a cliché, is "Use a better algorithm before falling back to assembly." No point in hand-crafting an assembly version when a smarter version could be whipped out in a higher-level language. Heck, with the right algorithm you could code a fast blitter with HyperTalk. Now

if you can just find that magic algorithm. Until then we'll work with C and small doses of assembly language.

Logical Masking

The next few examples of blitters will employ logical masking as their core technique. So before jumping into the blitters you should spend a few paragraphs reviewing how logical masking works.

For the masking to work you need a mask, the background, and the sprite. Let's use the first frame of our Tommy sprite. We'll need a background, so let's put Tommy in an environment he's familiar with, a pixelly brick background (see Figure 5-3).

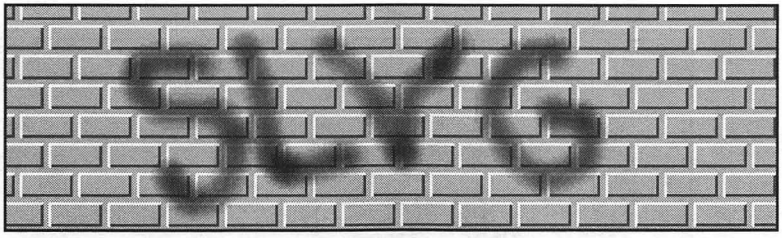


Figure 5-3. The 'hood

The steps for logical masking are easy and even fun. You get to use those nifty C logical bit operators like '&,' '|,' and my favorite, '~.'

- Step 1: Invert the mask (see Figure 5-4)

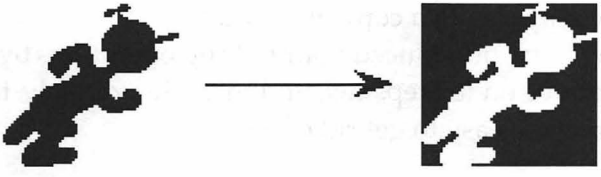


Figure 5-4. Invert the mask

- Step 2: Bit-wise 'and' the inverted mask with the background (Figure 5-5)

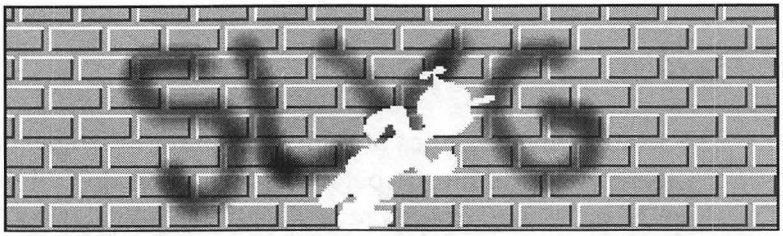


Figure 5-5. Inverted mask with background

- Step 3: Bit-wise 'or' sprite with the result of Step two (Figure 5-6)

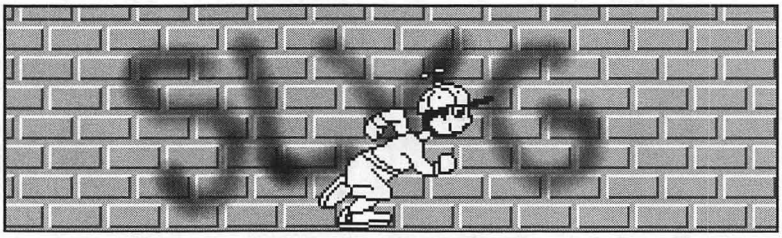


Figure 5-6. Tommy in the 'hood

After these three steps you'll have a sprite properly composited over the background.

As the quick-to-optimize among you noticed, Step one can be done as part of building the sprites and stored in that form. The only ones who will know that you skipped this step are you, me, and any player with a copy of ResEdit.

Now that you reduced your blitting operations by a third, it's time to move on to Steps two and three. Regrettably, these two steps aren't so easy to get rid of. Darn.

Brain Dead III

As a base platform let's take the BrainDeadBlitter and add logical masking. From here we'll start examining the areas that can be improved. Don't worry, there are lots of them.

```
void  BrainDeadLogicalMaskedBlit(  PixMapPtr   srcMap,
                                   PixMapPtr   dstMap,
                                   PixMapPtr   maskMap,
                                   Rect *      srcR,
                                   Rect *      dstR)
{
      char *      srcAddr;
      char *      dstAddr;
      char *      maskAddr;
      short       srcRowbytes;
      short       dstRowbytes;
      short       row, pixel;

      // Extract the start of the pixels out of the pixmap
      srcAddr = srcMap->baseAddr;
      dstAddr = dstMap->baseAddr;
      maskAddr = maskMap->baseAddr;

      // Extract the rowBytes out of the pixmap
      srcRowbytes = srcMap->rowBytes & Ox3FFF;
      dstRowbytes = dstMap->rowBytes & Ox3FFF;

      for(row = srcR->top; row <= srcR->bottom; row++)
      {
            char *      srcPtr;
            char *      dstPtr;
            char *      maskPtr;

            // Calculate the start of each row
            srcPtr = srcAddr +
                        (row * srcRowbytes + srcR->left);

            dstPtr = dstAddr +
                        (row * dstRowbytes + dstR->left);

            maskPtr = maskAddr +
                        (row * srcRowbytes + srcR->left);
```

```
                for(  pixel = srcR->left;
                      pixel <= srcR->right;
                      pixel++)
                {
                      char scratch;

                      // Combine the sprite on top of background
                      // using the mask to punch out a hole.

                      scratch = *dstPtr & *maskPtr;
                      scratch |= *srcPtr;
                      *dstPtr = scratch;

                      // March along to the next pixel
                      dstPtr++;
                      srcPtr++;
                      maskPtr++;
                }
          }
}
```

The major change from the previous version is the addition of the masking code. This chunk of code assumes that the mask matches the size, height, and rowBytes of the sprite pixel image. The mask is also wrapped up in a pixel map, with each byte colored white (0x00) or black(0xFF). As mentioned previously, the mask is expanded to a byte per pixel so that the code can march through the mask at the same stride that it moves through the source and destination pixels.

```
char scratch;

// Combine the sprite on top of background
// using the mask to punch out a hole.

scratch = *dstPtr & *maskPtr;
scratch |= *srcPtr;
*dstPtr = scratch;
```

These three lines of code are the C equivalent of the logical operations we performed on Tommy. First the pointer to the background is dereferenced to get at the background pixel. The

background pixel is then "and-ed" with the matching part of the mask. The resultant pixel is saved in a temporary variable. This step of the masking operation punches a hole in the background only where the sprite will appear opaque. The temporary variable that holds the mask and the background is then "or-ed" with the sprite pixel. The pixel in the temporary variable now holds what we need. The pixel that is referenced by the destination pointer is replaced with the fully composited pixel. If the destination was on-screen you would then see a one-pixel chunk of the sprite over the background.

Simple Optimizations

Time to start making this code run faster. The best place to optimize is obviously where the processor spends most of its time. And in the case of the brain dead blitter it's spending all of its waking hours in the loop that combines the mask and the sprite with the background. Anything that happens before the loop is inconsequential in comparison to the amount of cycles burned in the bowels of that "for" loop. For our first crack at optimization let's knock down the amount of time spent in that loop.

Movin' Large

The best way to reduce the amount of time in the loop is to increase the amount of work that is done per cycle of the loop. And that's easy to do with this loop. For each cycle of the loop only one composited pixel is produced. Since each pixel is a byte, the loop is working on a single byte at a time. This would be great if this code were running on an 8-bit processor, but it isn't. It's running on a 32-bit processor. The code should be working at the natural width of the processor, 32 bits, or a long word at a time. That's easy enough to arrange.

```
for(row = srcR->top; row <= srcR->bottom; row++)
{
      short longsPerRow;
      long *      srcPtr;
      long *      dstPtr;
      long *      maskPtr;

      // Calculate the start of each row
      srcPtr = srcAddr +
                  (row * srcRowbytes + srcR->left);

      dstPtr = dstAddr +
                  (row * dstRowbytes + dstR->left);

      maskPtr = maskAddr +
                  (row * srcRowbytes + srcR->left);

      longsPerRow = (srcR->right--srcR->left) / 4;

      for(pixel = 0; pixel <= longsPerRow; pixel++)
      {
            long scratch;

            // Combine the sprite on top of background
            // using the mask to punch out a hole.

            scratch = *dstPtr & *maskPtr;
            scratch |= *srcPtr;
            *dstPtr = scratch;

            // March along to the next 4 pixels.
            dstPtr++;
            srcPtr++;
            maskPtr++;
      }
}
```

Because you've changed the pointers that accessed the sprite, background, and mask into long pointers instead of char pointers, the innermost loop is now doing four times as much work per cycle as it was before. Now that the loop is moving four pixels at a time, the 'for' loop needs to loop four fewer times. This is accomplished by dividing the number of pixels per row by four. Which is great—by reducing the number of times you loop you also spend more time in the body of the loop and less time in the overhead code of the loop.

A fourfold speedup with a few changes in the code—that was easy. Too easy. The code now only works correctly for blits with pixel widths that are evenly divisible by four. Great if you're trying to blit 32 pixels per row, but if you're trying to blit 30 pixels across you'll end up only moving 28 pixels. Not a good thing for a blitter. You'd be pretty upset if `CopyBits` only bothered to blit 28 out of your 30 pixels. There are two ways to fix this bug. The easiest is to use the Sun Tzu approach and win the battle by not being there. Which if you somehow missed your Eastern Philosophy classes translates to only using sprites that have widths divisible by four. This isn't as ridiculous as its sounds. Most arcade games could easily be forced to follow the divisible-by-four constraint without affecting the game play. The other way is to ignore Mr. Tzu and head right into the battle and fix the bug.

```
for(row = srcR->top; row <= srcR->bottom; row++)
{
      short       longsPerRow;
      short       extraPixels;
      long *      srcPtr;
      long *      dstPtr;
      long *      maskPtr;

      char *      srcBytePtr;
      char *      dstBytePtr;
      char *      maskBytePtr;

      // Calculate the start of each row
      srcPtr = srcAddr +
                  (row * srcRowbytes + srcR->left);

      dstPtr = dstAddr +
                  (row * dstRowbytes + dstR->left);

      maskPtr = maskAddr +
                  (row * srcRowbytes + srcR->left);

      longsPerRow = (srcR->right--srcR->left) / 4;
      extraPixels = (srcR->right--srcR->left) -
                                    (longsPerRow * 4);

      for(pixel = 0; pixel <= longsPerRow; pixel++)
      {
```

```
            long scratch;

            // Combine the sprite on top of background
            // using the mask to punch out a hole.

            scratch = *dstPtr & *maskPtr;
            scratch |= *srcPtr;
            *dstPtr = scratch;

            // Skip over four pixels
            dstPtr++;
            srcPtr++;
            maskPtr++;
      }

      // set up the Byte pointers
      srcBytePtr = srcPtr;
      dstBytePtr = dstPtr;
      maskBytePtr = maskPtr;

      // Now copy over the extra 1, 2, or 3 pixels
      for(pixel = 0; pixel <= extraPixels; pixel++)
      {
            char scratch;

            // Combine the sprite on top of background
            // using the mask to punch out a hole.

            scratch = *dstBytePtr & *maskBytePtr;
            scratch |= *srcBytePtr;
            *dstBytePtr = scratch;

            // March along to the next pixel
            dstBytePtr++;
            srcBytePtr++;
            maskBytePtr++;
      }
}
```

To copy over the few extra bytes, which will be between one and three pixels, another loop is set up with byte pointers instead of the long pointers used previously. The loop looks like a sinkhole of cycles, but since it maxes out at three pixels its bark is worse than its byte. Ohhhh. You knew it had to happen. I held back for over two hundred pages. Anyway, the fourfold gain in moving to long

pointers more than makes up for the three extra bytes that might have to moved.

Movin' Code

The blitter is now masking out as many bytes at a time as the processor is capable of. A quick look at the remaining code shows several targets of opportunity. The first is what compiler writers refer to as an invariant. An invariant expression is a chunk of code that does not vary during the execution of a loop. A classic example is the following line from the blitter.

```
longsPerRow = (srcR->right--srcR->left) / 4;
extraPixels = (srcR->right--srcR->left) -
                                (longsPerRow * 4);
```

During the execution of the loop the number of long words per row or the number of extra pixels will not change. But each of these expressions will be executed for each row of pixels that is transferred. What a waste. Moving these invariant expressions to outside of the loop will speed up the blitter a slight amount. Not as great as a gain as moving to longs, but a cycle saved is a cycle earned. Good compilers will be able to spot most common invariants and move them for you. You trust your compiler, don't you? Right, neither do I. Let's move those wastes of time ourselves.

```
short  longsPerRow;
short  extraPixels;

longsPerRow = (srcR->right -- srcR->left) / 4;
extraPixels = (srcR->right -- srcR->left) -
                                    (longsPerRow * 4);

for(row = srcR->top; row <= srcR->bottom; row++)
{
      long *      srcPtr;
      long *      dstPtr;
      long *      maskPtr;

      char *      srcBytePtr;
```

```
        char *      dstBytePtr;
        char *      maskBytePtr;
                                .
                                .
                                .
```

The next target of optimization is where the address for the start of each row is calculated. For every row of pixels the same code is executed, producing a different address for each row. The invariant here is that the number of bytes between one row and the next remains constant. With this invariant we can recode the address calculations, a multiply and a subtract, to use only one add. A good rule of thumb is, if you see a multiply in a loop you are probably looking at an absolute calculation. And that absolute calculation can probably be replaced with an addition, making the calculation now a relative one.

```
// Calculate the start of each row
srcPtr = srcAddr;
dstPtr = dstAddr;
maskPtr = maskAddr;

// Find the byte offset from one row to the next
diffBetweenRows = srcRowbytes--(srcR->right--srcR->left);

for(row = srcR->top; row <= srcR->bottom; row++)
{
                                .
                                .
                                .
        dstPtr = dstBytePtr + diffBetweenRows;
        srcPtr = srcBytePtr + diffBetweenRows;
        maskPtr = maskBytePtr + diffBetweenRows;
}
```

Before the outermost row loop is started, the long pointers are initialized to point at the first line of the pixel map. Then the number of bytes from the end of one row to the start of the next is stashed away. After the pixels have been blitted and masked for the current row, the long pointers are set to the next rows by simply

adding the stashed value to the byte pointers, which conveniently enough are sitting at the last logical pixel of the row.

In each of the "for" loops the current index is checked against a value that is not going to change. These for loops can be replaced with the slightly more efficient "while" loops. And while we're going after microefficiencies, let's get rid of those scratch variables. They annoy me.

From this unsightly code

```
long scratch;

// Combine the sprite on top of background
// using the mask to punch out a hole.

scratch = *dstPtr & *maskPtr;
scratch |= *srcPtr;
*dstPtr = scratch;

// Skip over four pixels
dstPtr++;
srcPtr++;
maskPtr++;
```

to this. Notice the almost unreadable, condensed lines of code: the telltale sign that optimization has taken place.

```
*dstPtr = (*dstPtr & *maskPtr++) | *srcPtr++;
dstPtr++;
```

You might try to make this code a little more unreadable by doing this.

```
*dstPtr++ = (*dstPtr & *maskPtr++) | *srcPtr++;
```

Don't. It might work. It might not. It's strictly up to the compiler. If the compiler wants to, it can dereference the left side and then increment the pointer before doing anything on the right side. Which will leave the destination pointer staring at the wrong value. And this is the value that will be "and-ed" with the mask. You won't crash. You'll just end up with a very difficult-to-spot bug.

The mask and source pointer can be postincremented, as they are used only once within the statement.

After those last few hack-and-slash edits, the inner part of the blitter looks like this.

```
short       longsPerRow;
short       extraPixels;
short       diffBetweenRows;
short       rowCount;
short       row;
short       pixels;

long *      srcPtr;
long *      dstPtr;
long *      maskPtr;

longsPerRow = (srcR->right--srcR->left) / 4;
extraPixels = (srcR->right--srcR->left) -
                                        (longsPerRow * 4);

diffBetweenRows = srcRowbytes--(srcR->right--srcR->left);

// Calculate the start of each row
srcPtr = srcAddr;
dstPtr = dstAddr;
maskPtr = maskAddr;

row = rowCount = srcR->bottom--srcR->top;

while(row--)
{
      char *      srcBytePtr;
      char *      dstBytePtr;
      char *      maskBytePtr;

      pixels = longsPerRow;

      while(pixels--)
      {
            // Combine the sprite on top of background
            // using the mask to punch out a hole.

            *dstPtr = (*dstPtr & *maskPtr++) | *srcPtr++;
            dstPtr++;
      }
```

```
        // set up the Byte pointers
        srcBytePtr = srcPtr;
        dstBytePtr = dstPtr;
        maskBytePtr = maskPtr;

        // Copy over the extra 1, 2, or 3 pixels
        pixels = extraPixels;

        while(pixels--)
        {
                // Combine the sprite on top of background
                // using the mask to punch out a hole.

                *dstBytePtr = (*dstBytePtr & *maskBytePtr++) |
                                        *srcBytePtr++;
                dstBytePtr++;
        }

        dstPtr = dstBytePtr + diffBetweenRows;
        srcPtr = srcBytePtr + diffBetweenRows;
        maskPtr = maskBytePtr + diffBetweenRows;
}
```

You might feel compelled to start sprinkling "register" declarations all over this code. Don't. Modern compilers can usually do a better job of figuring out which variables should be placed into registers than you can. If in doubt, look at the assembly dumps with your register suggestions and the one your compiler produces. Remember to turn on the register coloring optimization for your compiler if you want this to be a fair comparison. If you can do a better job than the compiler reward yourself ten hacker points and start pasting "register" wherever you feel like it.

Movin' in Place

After you have a blitter working at the natural data size of the processor, in our case 32 bits wide, the next optimization step is alignment.

A long word is properly aligned if its starting address begins on a long word boundary. In our case the long's address must be evenly divisible by four, otherwise some of the long's bytes will lie

across a long word boundary. When this happens every access to the badly placed long made by the processor will have to be broken up into several smaller and slower memory fetches. And since you are moving a long at a time within the row, you'll be suffering with the misalignment you started with for every pixel you process. When your pointers are properly aligned you will have the processor working at its peak speed.

In the case of our blitter there are three pointers—the source, destination, and mask—that we need to worry about. The source and mask pointers reference memory blocks that don't change for the life of the sprite. Which makes them easy candidates for long word alignment. Simply force the alignment when they are initially created. Both the memory manager and 32-Bit QuickDraw allow you to create long word blocks of pixels. The memory manager always returns aligned blocks when you use `NewPtr` or `NewHandle`. Or by setting the proper align flag you can force your GWorlds to proper alignment. With the source and mask handily taken care of, that leaves the destination pointer as our only optimization candidate.

As the sprite moves across the screen the location of the destination pointer changes accordingly. Without forcing your sprite to move in four-pixel increments, you'll end up with a misaligned destination pointer for around one out of every four pixels. To handle alignment for the destination pointer you just need to follow the same plan of attack as when the blitter changed from a byte at a time to processing longs. First find out if the destination pointer is misaligned.

```
unalignedBytes = 4--((long)dstPtr & 3L );
```

If a long pointer is misaligned one or both of its least significant bits will be set. By masking all the bits above the lowest two, you end up with the number of bytes the pointer is off by.

```
if(unalignedBytes)
{
      srcBytePtr = srcPtr;
      dstBytePtr = dstPtr;
      maskBytePtr = maskPtr;

      while(unalignedBytes--)
      {
            *dstBytePtr = (*dstBytePtr & *maskBytePtr++) |
                                          *srcBytePtr++;
            dstBytePtr++;
      }

      extraPixels -= unalignedBytes;

      srcPtr = (long *) srcBytePtr;
      dstPtr = (long *) dstBytePtr;
      maskPtr = (long *) maskBytePtr;
}
```

If the pointer is misaligned this snippet of code copies over each misbehaving byte, masking as it goes. Once those few bytes have been blitted over, the destination pointer is long word aligned. After the loop is finished the number of misaligned pixels is subtracted from the number of extras pixels per row and the long pointers are reestablished.

```
while(pixels--)
{
      // Combine the sprite on top of background
      // using the mask to punch out a hole.

      *dstPtr = (*dstPtr & *maskPtr++) | *srcPtr++;
      dstPtr++;
}

// set up the Byte pointers
srcBytePtr = srcPtr;
dstBytePtr = dstPtr;
maskBytePtr = maskPtr;

// Copy over the extra 1, 2, or 3 pixels
pixels = extraPixels;

while(pixels--)
```

```
{
      // Combine the sprite on top of background
      // using the mask to punch out a hole.

      *dstBytePtr = (*dstBytePtr & *maskBytePtr++) |
                                          *srcBytePtr++;
      dstBytePtr++;
}
```

The rest of the blitter works as previously discussed with the only note relating to handling the extra pixels. After the blitter has run through all the full-sized longs in the row, it then drops down to blitting the few extra bytes at the end of the row. Because of the processing to force alignment, the number of pixels left at the end of the row might have changed. But as before you'll never be copying more than three bytes per row.

Unrolling Your Own

The last few optimizations have tuned up this chunk of code nicely. A quick scan with the OptimoMeter™ shows that we have yet to perform everyone's favorite code tuning technique: loop unrolling.

When you have a piece of code that spends most of its processing time in a loop as our blitter does, one sure way to increase execution speed is to spend more time in the loop's core code. Every loop you write spends some percentage of its time in code that handles the looping construct. And if the code is just counting off iterations, it's not doing the truly important stuff, like blitting our pixels. By unrolling a loop your code spends less time in the fat and more time in the meat. Let's look at our blitter as a prime place for some unrolling.

```
while(pixels--)
{
      // Combine the sprite on top of background
      // using the mask to punch out a hole.

      *dstPtr = (*dstPtr & *maskPtr++) | *srcPtr++;
      dstPtr++;
}
```

At the core of our blitter the code loops for the number of longs in each row. I'm ignoring the alignment loops since for average-sized sprites most of the blitting time is spent in this loop. For each long word processed the looping code must test if the "pixels" variable has been depleted, and if not it is then decremented by one. This extra processing done for loop is straight overhead. Loop unrolling attempts to reduce the amount of overhead by doing more work for each loop iteration. Here is the blitting loop unrolled slightly.

```
pixels /= 4;

while(pixels--)
{
      // Combine the sprite on top of background
      // using the mask to punch out a hole.

      *dstPtr = (*dstPtr & *maskPtr++) | *srcPtr++;
      dstPtr++;

      *dstPtr = (*dstPtr & *maskPtr++) | *srcPtr++;
      dstPtr++;

      *dstPtr = (*dstPtr & *maskPtr++) | *srcPtr++;
      dstPtr++;

      *dstPtr = (*dstPtr & *maskPtr++) | *srcPtr++;
      dstPtr++;
}
```

Inside the loop the blitting code has been duplicated or unrolled four times. To match the amount of unrolling performed, the loop counter has been reduced by four before entering the loop. Now the loop is processing four times as much data for each loop test. Net result is usually faster code.

Remembering that TANSTAFL (there ain't no such thing as free lunch) is a primal force; you need to look at what you lost in the trade for speed. Memory, for starters. This blitting loop compiles down to code that is around four times larger than the original version. The next trade-off is generality. The original loop worked for all row sizes. This unrolled version will only work correctly when

the number of longs to be blitted is divisible by the number of times the loop was unrolled. Déjà vu. This is the same problem the code had when it was updated to blit whole long words instead of bytes. And you can apply the technique. Loop for the whole amounts and then pick up the remainders at the end.

```
#define kUnrolledAmt    4

long counter = pixels / kUnrolledAmt;

while(counter--)
{
      // Combine the sprite on top of background
      // using the mask to punch out a hole.

      *dstPtr = (*dstPtr & *maskPtr++) | *srcPtr++;
      dstPtr++;

      *dstPtr = (*dstPtr & *maskPtr++) | *srcPtr++;
      dstPtr++;

      *dstPtr = (*dstPtr & *maskPtr++) | *srcPtr++;
      dstPtr++;

      *dstPtr = (*dstPtr & *maskPtr++) | *srcPtr++;
      dstPtr++;
}

// Copy over the longs left over
counter = pixels--((pixels / kUnrolledAmt) * kUnrolledAmt);

while(counter--)
{
      *dstPtr = (*dstPtr & *maskPtr++) | *srcPtr++;
      dstPtr++;
}
```

What if you'd like to increase the amount of unrolling for this loop? You'd have to unroll the correct number of lines and, after changing the proper constant, recompile the code. Not exactly the most efficient way of tuning to different-sized sprites. What you would like is a piece of code that has the speed benefits of unrolled loops without the loss in generality. Thankfully, someone else had

the exact same dilemma. Digging through my archive of "Great E-mails of American History" I uncovered this beauty.

```
>From research!ucbvax!dagobah!td Sun Nov 13 07:35:46
1983
Received: by ucbvax.ARPA (4.16/4.13)
Received: by dagobah.LFL (4.6/4.6b)
Date: Thu, 10 Nov 83 17:57:56 PST
From: ucbvax!dagobah!td (Tom Duff)
Message-Id: <8311110157.AA01034@dagobah.LFL>
To: ucbvax!decvax!hcr!rrg, ucbvax!ihnp4!hcr!rrg,
ucbvax!research!dmr, ucbvax!research!rob

Consider the following routine, abstracted from code
which copies an array of shorts into the Programmed IO
data register of an Evans & Sutherland Picture System II:

send(to, from, count)
register short *to, *from;
register count;
{
      do
            *to = *from++;
      while(--count>0);
}

(Obviously, this fails if the count is zero.) The VAX C
compiler compiles the loop into 2 instructions (a movw
and a sobleq, I think.) As it turns out, this loop was
the bottleneck in a real-time animation playback pro-
gram which ran too slowly by about 50%. The standard
way to get more speed out of something like this is to
unwind the loop a few times, decreasing the number of
sobleqs. When you do that, you wind up with a leftover
partial loop. I usually handle this in C with a switch
that indexes a list of copies of the original loop
body. Of course, if I were writing assembly language
code, I'd just jump into the middle of the unwound loop
to deal with the leftovers. Thinking about this yester-
day, the following implementation occurred to me:

send(to, from, count)
register short *to, *from;
register count;
```

```
{
        register n=(count+7)/8;
        switch(count%8){
          case 0:  do{     *to = *from++;
          case 7:          *to = *from++;
          case 6:          *to = *from++;
          case 5:          *to = *from++;
          case 4:          *to = *from++;
          case 3:          *to = *from++;
          case 2:          *to = *from++;
          case 1:          *to = *from++;
                   }while(--n > 0);
        }
}
```

Disgusting, no? But it compiles and runs just fine. I feel a combination of pride and revulsion at this discovery. If no one's thought of it before, I think I'll name it after myself.

It amazes me that after 10 years of writing C there are still little corners that I haven't explored fully. (Actually, I have another revolting way to use switches to implement interrupt driven state machines but it's too horrid to go into.)

Many people (even bwk?) have said that the worst feature of C is that switches don't break automatically before each case label. This code forms some sort of argument in that debate, but I'm not sure whether it's for or against.

yrs trly
Tom

Thanks a million, Tom, that was just the thing our blitter needed and provided proof that my Internet account is tax-deductible. A few cut and pastes later our blitter is now sporting the best, if not the most obscure, that C has to offer.

```
#define kUnrolledAmt 8

switch( pixels % kUnrolledAmt )
{
      case 0:
      do
      {
            *dstPtr = (*dstPtr & *maskPtr++) | *srcPtr++;
            dstPtr++;
      case 7:
            *dstPtr = (*dstPtr & *maskPtr++) | *srcPtr++;
            dstPtr++;
      case 6:
            *dstPtr = (*dstPtr & *maskPtr++) | *srcPtr++;
            dstPtr++;
      case 5:
            *dstPtr = (*dstPtr & *maskPtr++) | *srcPtr++;
            dstPtr++;
      case 4:
            *dstPtr = (*dstPtr & *maskPtr++) | *srcPtr++;
            dstPtr++;
      case 3:
            *dstPtr = (*dstPtr & *maskPtr++) | *srcPtr++;
            dstPtr++;
      case 2:
            *dstPtr = (*dstPtr & *maskPtr++) | *srcPtr++;
            dstPtr++;
      case 1:
            *dstPtr = (*dstPtr & *maskPtr++) | *srcPtr++;
            dstPtr++;

      } while( --pixels );
}
```

After that last round of surgery the blitter has an unrolled inner core that will transfer 32 pixels for each iteration of the loop. It's got the benefits of assembly programming with the same level of readability. Isn't C a wonderful language? Makes you wonder what that interrupt-state machine code looks like. By the way, since no one else claimed to have invented this exotic code morsel before Tom, he made good on his threat and named it after himself: "Duff's Device."

Run length masking

Now that you've seen the parade of brute-force methods of blitting images, let's look over a couple of more elegant solutions than just looping and slamming pixels around.

The first method that'll come under our code microscope is what I call "Run Length Masking." The general idea is to encode the mask in such a way that the blitting code can quickly copy over only the pixels of the source image that need to be copied. The previous blitting examples moved over every pixel of the sprite regardless of whether all those pixels would need to be displayed. If the pixels aren't going to end up on the screen, then you're wasting time for each one of those masked pixels.

When `CopyBits` is passed a mask region, it goes on to copy only the pixels that lie within the region and ignores the rest. That's exactly what you want to do, but faster. Digging out those handy guidelines on how to be faster than `CopyBits`, you can see that you just need to program your blitter in a less general manner. Easy; let's use the same constraints as last time.

←⊇Eight bit indexed color

←⊇Matching color tables

←⊇Source and destination areas are the same size

For even greater speed (and to avoid potential lawsuits), the mask needs to be encoded into something other than a region and yet still provide the ability to determine quickly that you have a run of pixels. What would such an encoding look like? Interestingly enough, it could look like an ordinary bitmap.

If you make a call to `GetCIcon` you'll get handed back a `CIconHandle`. This handle will point to a `CIcon` structure that holds all the goodies that make up a color icon. First off the `CIcon` structure contains a pixel map that contains all the color pixels that comprise the full-color version of the icon. After the color portion of the structure comes the mask for the icon. The mask is used just like a region mask in `CopyBits`, to discriminate which pixels get copied from the color

icon's pixel map and which don't. Unlike `CopyBits` this mask is encoded as a bitmap, not as a region. You've seen bitmaps used as masks before with the Toolbox routine `CopyMask`, so this is nothing new. With `CopyMask`, a bit set in the bitmap indicates that the corresponding pixel in the pixel map should be copied and when a bit is cleared the matching pixel stays where it lies. So what does this all have to do with finding the runs of pixels that need to be blitted? Plenty.

The trick to finding a run of pixels is to look at a bitmap not as a collection of individual bits. But to look at it instead as an array of larger ordinal types, like bytes—shorts or longs. When viewed from this angle, finding a run of pixels to copy is as easy as using a switch statement.

As an example, let's imagine that you have a color icon that is only one line high and eight pixels across. With only eight color pixels in the source image the mask bitmap will consist of only eight bits or a single byte (for this example I'm ignoring the `rowBytes` padding). Now if you compare that byte containing all eight mask bits against 0xFF you can determine quickly if you have run of eight consecutive pixels. Or you could perform the inverse test and compare the byte mask against 0 to see if you need to bother copying any pixels at all.

Between 0 and 0xFF are 254 other combinations of pixel runs—perfect for a switch statement. For each byte in the mask, you would have your blitter switch to the appropriate mask value.

```
switch(maskByte)
{
      case 0x0: // no need to do anything
      break;

      case 0x1: // b0000 0001: copy the last pixel
      break;

      case 0x2: // b0000 0010: copy the second to last pixel
      break;
                        •
                        •
                        •
                        •
```

```
        case 0xFE // b1111 1110: copy all pixels but last
        break;

        case 0xFF // copy all eight pixels
        break;
}
```

Once the switch statement has branched to one of the mask value labels you'll want to start copying those pixels. And while copying the pixels you'll want to take full advantage of any consecutive runs of pixels indicated by the mask. This is easy enough to do if you switch from thinking of the mask as a byte and back to thinking of it as a collections of bits. As a collection of bits the mask becomes a recipe for how most efficiently to copy only the pixels that need copying.

Again, an example will help. Say your switch statement has been handed a mask value of 111, or 0x6F. As a collection of bits, this byte looks like b0110 1111. The binary-encoded blitting recipe reads as follows: Skip the first pixel; copy the next two pixels; finish off by copying the last four pixels in one long gulp. This recipe transcribed for C would look like the following (assuming, of course, that you have source and destination pointers).

```
// srcPtr is char * into the icon's color pixels
// dstPtr is another char * into the destination
// offscreen

switch(maskByte)
{
      case 0x0: // no need to do anything
            break;

      case 0x1: // b0000 0001: copy the last pixel
            *(dstPtr + 7) = *(srcPtr + 7);
            break;

      case 0x2: // b0000 0010: copy the second to last pixel
            *(dstPtr + 6) = *(srcPtr + 6);
            break;
```

```
                                .
                                .
                                .
        case 0x6F: // b0110 1111
              // skip the first pixel and copy two
              *(short *)(dstPtr + 1) = *(short *)(srcPtr + 1);
              *(long *)(dstPtr + 4) = *(long *)(srcPtr + 4);
                                .
                                .
                                .
        case 0xFE // b1111 1110: copy all pixels but last
              *(long *)(dstPtr) = *(long *)(srcPtr);
              *(short *)(dstPtr + 4) = *(short *)(srcPtr + 4);
              *(dstPtr + 6) = *(srcPtr + 6);
              break;
        case 0xFF // copy all eight pixels
              *(long *)(dstPtr) = *(long *)(srcPtr);
              *(long *)(dstPtr + 4) = *(long *)(srcPtr + 4);
              break;
}
```

Each possible value of the mask is broken down into the minimum number of bytes, short or long transfers needed to blit the pixels from the source into the destination offscreen. This is a pretty efficient way to blit pixels. To copy an eight-pixel run of pixels with the previous blitter would have required eight separate masking operations. Here you only needed two long transfers. And for a case like 0x6F, the previous blitter would have wasted time processing two pixels that will never be seen. With this technique, you transfer only the pixels that are needed.

For our example blitter, let's write the transfer portion to use only the lower nybble of the mask byte. This reduction will give the switch only fifteen different blitting combinations to worry about and save a few pages of code. This reduction also makes this function a good candidate for inlining.

```
void  BlitFourPixelRun( char * srcAddr,
                        char * dstAddr,
                        char runMask)
{
      // Use only the lower nybble of the mask.
      // One nybble equals four 8 bit pixels
      switch(runMask & 0x0F)
      {
            case 1: // b0001, skip first 3 pixels copy last
                  *(dstAddr + 3) = *(srcAddr +3);
                  break;

            case 2: // b0010, skip over 2 pixels
                  *(dstAddr + 2) = *(srcAddr + 2);
                  break;

            case 3: // b0011, copy a shorts worth of pixels
                  *(short *)(dstAddr + 2) =
                              *(short *)(srcAddr + 2);
                  break;

            case 4: // b0100, need to skip first pixel
                  *(dstAddr + 1) = *(srcAddr + 1);
                  break;

            case 5: // b0101, skip first, copy second,
                        // skip third, copy last
                  *(dstAddr + 1) = *(srcAddr + 1);
                  *(dstAddr + 3) = *(srcAddr + 3);
                  break;

            case 6: // b0110, copy another shorts worth
                  *(short *)(dstAddr + 1) =
                              *(short *)(srcAddr + 1);
                  break;

            case 7: // b0111, copy a short then a byte
                  *(short *)(dstAddr + 1) =
                                    *(short *)(srcAddr + 1);
                  *(dstAddr + 3) = *(srcAddr + 3);
                  break;

            case 8: //b1000, copy the first byte
```

```
                *dstAddr = *srcAddr;
                break;

            case 9: //b1001, copy first and last byte
                *dstAddr = *srcAddr;
                *(dstAddr + 3) = *(srcAddr + 3);
                break;

            case 10: // b1010, copy first and second to last
                *dstAddr = *srcAddr;
                *(dstAddr + 2) = *(srcAddr + 2);
                break;

            case 11: //b1011, copy first byte
                        // and a short at the end
                *dstAddr = *srcAddr;
                *(short *)(dstAddr + 2) =
                                *(short *)(srcAddr + 2);
                 break;

            case 12: //b1100, copy first short
                *(short *)dstAddr = *(short *)srcAddr;
                break;

            case 13: //b1101, copy first short and last byte
                *(short *) dstAddr = *(short *)srcAddr;
                *(dstAddr + 3) = *(srcAddr + 3);
                break;

            case 14: //b1110, copy first short and next byte
                *(short *)dstAddr = *(short *)srcAddr;
                *(dstAddr + 1) = *(srcAddr + 1);
                break;

            case 15: // b1111 copy a whole long
                *(long *)dstAddr = *(long *)srcAddr;
                break;
        }
    }
```

The function `BlitFourPixelRun` is no different from the previous examples, except that the masking of the upper nybble of the mask byte is part of the switch statement. With that upper nybble gone there are only fifteen tiny cases to worry about. If you want to avoid the performance penalty of calling `BlitFourPixelRun` twice, you can always code up a `BlitEightPixelRun` that handles all 255 cases.

A blitter that uses `BlitFourPixelRun` is not very different from all of the other blitters presented. With the one presented here, `RLEBlitter`, you pass in a pixel map pointer for the source and destination and a bitmap pointer to the 1-bit deep mask. This example assumes that source and destination bounds match and that you are blitting the full source pixel image.

```
void  RLEBlitter( PixMapPtr   srcMap,
                  PixMapPtr   dstMap,
                  BitMap *    maskMap,
                  Rect *      srcR,
                  Rect *      dstR)
{
      char *      srcAddr;
      char *      dstAddr;
      char *      maskAddr;
      short srcRowbytes;
      short dstRowbytes;
      short maskRowbytes;
      short row;
      short maskCnt;

      // Extract the start of the pixels out of the pixmap
      srcAddr = srcMap->baseAddr;
      dstAddr = dstMap->baseAddr;
      maskAddr = maskMap->baseAddr;

      // Extract the rowBytes out of the maps
      srcRowbytes = srcMap->rowBytes & 0x3FFF;
      dstRowbytes = dstMap->rowBytes & 0x3FFF;
      maskRowbytes = maskMap->rowBytes & 0x3FFF;

      for(row = srcR->top; row <= srcR->bottom; row++)
      {
```

```
            // The copying of each row of pixels is
            // controlled by the number
            // of bytes in the mask (including any
            // extra due to rowBytes padding).
            maskCnt = maskRowbytes;

            while(maskCnt--)
            {
                  // Copy the first four pixels of
                  // the mask's run
                  BlitFourPixelRun(srcAddr,
                                   dstAddr,
                                   (*maskAddr) >> 4);

                  // Copy the next four pixels of the run
                  // Get pointers to the next four pixels
                  srcAddr += 4;
                  dstAddr += 4;

                  // No need to shift the mask this time
                  BlitFourPixelRun(srcAddr,
                                   dstAddr,
                                   *maskAddr);

                  maskAddr++;
                  srcAddr += 4;
                  dstAddr += 4;
            }

      // Move the source and destination pointers to
      // the beginning of the next row of pixels to blit

      // The mask pointer will already be
      // pointing at its next row
            srcAddr =
                  srcMap->baseAddr + (row * srcRowbytes);

            dstAddr = dstMap->baseAddr +
                        (row * dstRowbytes + dstR->left);

      }
}
```

After setting up all the appropriate pointers, `RLEBlitter` falls into the usual blitter habit of cycling through all of the pixel rows contained in the source. Inside this loop is where each row is blitted across with two calls to `BlitFourPixelRun` for each byte contained in the mask's row of bits. After each call of `BlitFourPixelRun`, the source and destination pointers are moved over four bytes to account for the four pixels just transferred. On the second completion of `BlitFourPixelRun`, the mask's pointer is then bumped up to the next byte of the mask's row. This inner loop exits when all of the bytes with the row have been processed. At this point the source and destination pointers are adjusted so that they point to the next row of pixels to be moved. There is no need to adjust the mask pointer as it is already pointing at its next row—one of the benefits of using `rowBytes` as the inner loop's control variable.

What happens when the mask's `rowBytes` width does not match a natural number of pixels in the source pixel map; for instance, when the source image is 33 pixels across? In this case, the `rowBytes` of the mask's bitmap would have to be the largest even number of bytes that could hold 33 bits. Which, of course, would be six bytes containing 48 bits or 15 bits too many. The way the blitter is currently coded, the inner loop would cycle through the second-to-last byte just to handle that thirty-third bit contained in the mask and then continue on to waste time coping with the padding byte. Now there aren't any pixels in the source image beyond the thirty-third pixel. So what happens? Nothing, if those padding bits in the mask are cleared. With those bits clear, the calls to `BlitFourPixelRun` return without transferring any pixels at all. The worst that happens is that a couple of extra calls are made to `BlitFourPixelRun` without anything to show for it.

If these few extra calls bother you, you could fix the problem the easy way by defining your source images so that their width is an even number of bytes. Or you could be a little more adventurous and recode the blitter so that the mask is not encoded as a bitmap but as an array of bytes. In the previous example this would remove the extra padding byte at the end and the subsequent call to `BlitFourPixelRun`.

Sprite Compiling

Our blitting tour ends here with the most esoteric version of blitting, sprite compiling. Sprite compiling does exactly what it says, it compiles a sprite. The data from the sprite is compiled into machine code that when executed draws the sprite's image into memory. Cool, huh?

Don't be too intimidated by the compiler part of this blitter. There are no Backus-Naur descriptions, no state machines, no LEX or YACCs to master; the sprite data is simply converted into the assembly instructions a scan line at a time. *Sprite macros* would be a much more accurate term, but nowhere near as impressive-sounding.

There are two techniques to sprite compiling, generally referred to as *sprite compiling* and *mask compiling*. The first takes the sprite's source and mask data and produces machine code that contains the source image along with the assembly instructions to copy that data into memory. Code is only generated where the mask image says to do so. After the sprite has been compiled, its source and mask data, having been cemented into the code, are no longer needed. Mask compiling encodes only the masking information as code. The sprite's source image data is passed to the compiled mask code just like any other parameter. Speaking of parameters, both types of compiled sprites need to be passed a pointer to the sprites' final destination.

Decisions, decisions. Given two sprite-compiling methodologies which one to choose? It depends on the host platform. With the sprite's image data compiled directly into the code, the processor can only move data as fast as its immediate mode will allow. In the case of the 68K family, the immediate mode is limited to moving a long at a time, while other, better-endowed processors might have data movement instructions that can blast away data at rates far faster than our beloved 68K. Luckily, what the 68K family lacks in immediate data movement it more than makes up for in its ability to indirectly move data. Given a couple of pointers the 68K must have a gajillion ways of moving the data referenced by those pointers. And a couple of those instructions are barn-burners, much faster than the immediate modes offered by our host. Since the

mask compiler uses a pointer to the source image and another to the destination pixel map, and has no need for immediate data instructions, your choice of compiler has been preordained.

Compiling a Mask

Your strategy for compiling a sprite's mask starts by expanding the mask into an offscreen at a depth of eight bits per pixel. Then create a resizable block of memory to hold the compiled code. Generate the prologue code for the compiled sprite. Loop through each scan line of the mask, producing the code that will blit over the pixels. After all the lines of the mask have been exhausted, the cleanup code is appended to the end of the block. At the end of this process the block of memory will hold a code block just as if it had come from your favorite C compiler. By creating a function pointer referencing this freshly compiled sprite, you can blit that sprite anywhere you want. Here is the prototype for the function that you will be compiling.

```
void  CompiledBlitter( long srcPitch, // bytes to next row
                       long destPitch, // ditto
                       Ptr srcPtr, // Sprites image data
                       Ptr destPtr); // Dest. pixel map
```

The compiled code is built around this prototype. It has to know what the parameters are and in what order they will be passed. And failure to do so will cause your craftily compiled code to promptly explode. Keeping that in mind let's head off to build a compiler.

Moving Fast and Large

Within the core of the compiler each scan line will be broken down into runs of consecutive pixels. The compiler will then want to encode these runs using the most efficient transfer modes the processor has to offer. And just what might those be?

In the previous blitters presented you were stuck with moving a long at a time due to the logical masking. Without this constraint in

your way you can use the widest instructions possible to copy over the pixel runs. Without a background in 680x0 esoterica you would probably produce a compiler that generated assembly sequences like this.

```
movea.l (a0), (a1) ; copy the four source pixels to dest
```

Closed-captioning for the assembly-impaired: This code dereferences the pointer stashed in register `a0` and copies the long found there into the long pointed at by the register `a1`.

While this will work, there is a better instruction: `movem.l`. If you disassemble almost any function you have written you will see `movem.l` used, mainly to set up and restore the stack, like so.

```
movem.l      d3-d5/a2-a4,-(sp)      // save off registers
// lots of neat code utilizing the registers saved
movem.l      (sp)+,d3-d5/a2-a4      // restore registers
```

This nifty instruction can take a list of registers and copy their contents to the location pointed at by the destination pointer (in this example, the stack pointer, `sp`). This instruction can even be convinced to pre- or postincrement and decrement the destination pointer. What more could you want? An inverse instruction? You got it. The same instruction can take a pointer as the source and copy the data into the supplied destination registers. By combining these two variations you can get two lines of assembly code that can move 13 longs at a time.

```
movem.l   (a0), d0-d7/a2-a6
movem.l   d0-d7/a2-a6, a1
```

Only 13 longs, as registers `a0` and `a1` hold the source and destination and the `a7` is the stack pointer. Still, 13 longs moved with only two instructions is a heap better than 13 `movea.l`. In case

you're wondering, it is assumed that a5, and the a5 world it references, has been safely stashed away.

With a full arsenal of assembly instructions in our kit bag, it's now time to head off and build our mask compiler.

> On 68040 processors and greater, Motorola added the Move16 instruction, which will move 16 bytes without disturbing any registers at all, and it does it blazingly fast. The problem is that to use this instruction your source and destination pointers must line up on 16-byte boundaries. The other problem is that the darn thing is only available on '040 or above. That limits its use to those machines only, which are probably not the machines where you could truly use this added horsepower. Now if Motorola had only retrofitted all

Blitter Prologue

Just like any other compiler, our mask compiler must generate a function prologue before generating the meat of the function. The purpose of any function prologue is to save off the registers used by the function, set up the stack space for any automatic variables declared by the function, and load up any parameters that are passed in to their appropriate register locations.

```
void  CompiledBlitter( long srcPitch, // bytes to next row
                       long destPitch, // ditto
                       Ptr srcPtr, // Sprites image data
                       Ptr destPtr); // Dest. pixel map
```

First off, you want to move the four parameters over to their appropriate registers. Our compiler is hard-coded with the idea that the source RowBytes value and its associated pixel image pointer will be loaded into d0 and a0, respectively. Same for the destination arguments, except they'll reside in d1 and a1.

Following the C calling convention, the calling function will have erected a call frame by pushing the parameter in right-to-left order, with destPtr pushed first and ending with srcPitch. Before executing our compiled function the caller will push the re-

Table 5-1. Stack Frame	
dstPtr	
srcPtr	
dstPitch	
srcPitch	
Return Address	← SP

turn address on the stack, producing the stack frame illustrated in Table 5-1.

With one quick stroke of the `movem.l` brush, the entire stack is broken up and loaded into the proper registers. The offset of four in the `4(a7)` is used to skip over the return address.

```
movem.l 4(a7), d0-d1/a0-a1
```

Normally I would have declared our blitter prototype in reverse order from that shown. I like to have pointers listed first followed by any other auxiliary data. But then I couldn't have used `movem.l`, as it requires that data registers precede address registers, and I would have been forced to use four ordinary `move.l`.

After copying over the parameters the prologue needs to stash away any registers that the body of the code will use. The values stored in these registers will be restored to their appropriate registers at the end of the function. Again, `movem.l` will be used, but this time the way you would normally see it in the wild.

```
movem.l d3-d7/a2-a6,-(a7)
```

With d0, d1, a0, and a1 busy holding our needed parameters and the stack pointer (register a7) off doing what stack pointers do, the compiler is left with 10 registers to play with. The registers that hold our arguments don't have to be saved. These four registers are declared scratch registers and their values do not have to be saved across function calls. The enforcement for this policy is strictly vol-

untary; if you're feeling paranoid you can go ahead and save these registers.

Two lines of assembly is all it takes to set up our function's prologue, and neither line is tied to the parameters used by our function. Great. This means that these two lines can be hard-coded as defines and you can start building the mask compiler.

Our compiler takes two pixel map pointers, one to the source image and the other to the mask. The resultant compiled sprite is returned in the handle passed by the caller (the compilee?). The caller is responsible for allocating a handle for the generated code. In the case of an error, which could only happen if there isn't enough memory, the compiler will return an OS error.

```
// Hex for movem.l  4(a7), d0-d1/a0-a1    ; 4CEF 0303 0004
#define kCopyParametersPart1  0x4CEF0303
#define kCopyParametersPart2  0x0004

// Hex for movem.l   d3-d7/a2-a6,-(a7) ; 48E7 1F3E
#define kSaveRegisters  0x48E71F3E

#define kMaxInstructionLength 8     // 8 bytes max. length
```

The first two defines hold the hex values for copying the parameters followed by the hex code for saving off our needed registers. The hex codes were found by disassembling the code in Macsbug after creating a bogus function in an existing program (the Symantec bull's-eye tutorial works well) using the inline assembler. Turning on Macsbug's logging output will help here or TMONs playmen feature.

```
OSErr CompileSpriteMask(const PixMap *   srcPixels,
                        const PixMap *   maskPixels,
                        Handle           compiledMaskH)
{
      OSErr              err;

      *compiledMaskH = nil;

      ASSERT(srcPixels != nil);
      ASSERT(maskPixels != nil);
      ASSERT(compiledMaskH != nil);
```

```
        ASSERT(EqualRect( &srcPixels->bounds,
                                    &maskPixels->bounds));
```

A safety check is performed to verify that the caller didn't try to slip us any bogus data and, most important, that the source and mask images are the same size. After this check the source pointer is no longer needed. You could drop passing the source pointer all together. I kept it in to try stop any bonehead 3 A.M. mistakes I might make.

```
if(compiledMaskH)
{
        unsigned char*    codePtr;
        long rowCount;
        long maskRowBytes;
        long startingSize;

        rowCount = maskPixels->bounds.bottom -
                                maskPixels->bounds.top;
        maskRowBytes = maskPixels->rowBytes & 0x7FFF;

        //Make the handle as large as possible
        startingSize = rowCount * (maskRowBytes *
                                kMaxInstructionLength);
        SetHandleSize(compiledMaskH, startingSize);
        err = MemError();
        if(err == noErr)
```

The handle that will hold our crispy fresh code needs to be set to a size large enough to hold any amount of code that the mask might direct us to deliver. The handle will be trimmed back after all the processing is finished, when the actual size of the compiled code is known. For now, you just want the largest size it could possibly be. Which equates to the worst case of a mask without any consecutive runs of pixels (picture a mask that is a 50 percent gray pattern). This would require that each byte be moved individually, and the instructions needed to do that are a maximum of eight bytes long. Multiply this maximum by the number of rows and again by the number of bytes per row and you get the largest code size possible.

If you're lucky enough to have that kind of memory lying around, the code then heads off to write out the blitter's prologue.

```
{
      long scanLine;

      // Got the memory. Start compiling,
      // starting with the prologue
      HLock(compiledMaskH);
      codePtr = *compiledMaskH;

      *(unsigned long *)codePtr =
                        kCopyParametersPart1;
      ((unsigned long *)codePtr)++;

      *(unsigned short *)codePtr =
                        kCopyParametersPart2;
      ((unsigned short*)codePtr)++;
```

After locking down the handle and grabbing a pointer to the output handle, the code starts writing out the prologue. The code pointer is dereferenced after being cast to the size of the partial instruction being written. The pointer is then advanced by the size of that same partial instruction. Repeating the same process for the second part of our first `movem.l` completes the first line of the prologue.

```
//Stash away registers
*(unsigned long *)codePtr = kSaveRegisters;
((unsigned long*)codePtr)++;
```

The second `movem.l` is written out in the same manner as the first: write out the hex value for the wanted assembly instruction, bump the pointer up. There. The prologue has been written out successfully. And you thought this compiling business was tricky.

Blitter Core

The core of the mask blitter works a scan line at a time. The outermost loop simply loops though each scan line in the mask. From within each scan line consecutive runs of pixels will be extracted

and then compiled. Our compiler will take each run of pixels and break it down into a series of consecutively smaller `movem.l` instructions until the pixel run is down to four pixels. At that point the compiler will switch from using `movem.l` to the more pedestrian `move.l`. After the long's worth of pixels are exhausted, the compiler will drop down to moving two pixels at a time and finally finishes off moving a single pixel. This scenario assumes that the run starts longer than four pixels. Otherwise the compiler starts working with `move.l` and winds down from there.

```
// For each scanline in the mask compile it
for (scanLine = 0; scanLine < rowCount; scanLine++)
{
      EncodeScanLine(&codePtr, maskPixels, maskRowBytes);
      maskPixels += maskRowBytes;

      // Need to move the src and dest pointers
      // to the next scan line
      if (scanLine != rowCount--1)
      {
            //D1C0              ADDA.L    D0,A0
            *(unsigned short*)codePtr = 0xD1C0;
            ((unsigned short*)codePtr)++;

            //D3C1              ADDA.L    D1,A1
            *(unsigned short*)codePtr = 0xD3C1;
            ((unsigned short*)codePtr)++;
      }
}
```

The code that loops around the scan lines is straightforward enough; start with the mask pointer at the first scan line in the mask, finish at the last scan line, and in between compile the current scan line. After the scan line is compiled move the mask pointer to the next row in the mask pixel image. And since the compiler is generating code for each scan line, it needs to generate code for moving the source and destination pixel pointers to their next rows. This is easily accomplished by hard-coding a couple of `adda.l` into the output stream. In order to save a few cycles the compiler skips the bumping of the row pointers for the last scan

line in the mask. It would do no harm to do so, other than wasting time. Obviously, the real fun is inside `EncodeScanLine`.

```
void EncodeScanLine(unsigned char ** originalCodePtr,
                        const unsigned char * maskPixels,
                                    long maskRowBytes);
```

The core of our compiler, `EncodeScanLine` takes a pointer to our output stream, a pointer to the scan line it is about to compile and also determines how many bytes that scanline contains. The output stream pointer is passed by reference so that it can be updated to reflect its new position after all of the code has been streamed out. Before diving into the code generation, let's get an idea of how the runs are extracted from each line.

```
void EncodeScanLine(unsigned char ** originalCodePtr,
                        const unsigned char * maskPixels,
                                    long maskRowBytes)
{
      long              pixelsInLine;
      unsigned short    runStart;
      unsigned char     *codePtr = *originalCodePtr;

      runStart = 0;
      pixelsInLine = maskRowBytes;

      while (pixelsInLine)
      {
            long pixelsInRun = 0;

            // Find a pixel run
            while (pixelsInLine && *maskScanPixelP)
            {
                  maskScanPixelP++;
                  pixelsInRun++;
                  pixelsInLine--;
            }

            // compile the run
            while (pixelsInRun)
            {
                  EncodeRun(&codePtr,
```

```
                                &runStart,
                                &pixelsInRun);
            }

        // scan over transparent pixels
            while (pixelsInLine && !*maskScanPixelP)
            {
                runStart++;
                maskScanPixelP++;
                pixelsInLine--;
            }
        }

        // Update the caller's code Ptr
        *originalCodePtr = codePtr;
}
```

For ease of access the code stream reference is assigned to a local pointer. I hate typing "**" everywhere. The variable `runStart`, which keeps track of how many pixels from the beginning of the scan line the pixel run starts at, is initialized to zero, while `pixelsInLine` is set to the rowBytes of the mask's scan line.

The first loop is controlled by the number of pixels in the scan line. The code inside this loop counts down the variable `pixelsInLine` while it looks for pixel runs. When the guts of the loop is done it will have knocked `pixelsInLine` down to zero, and that scan line will have been transformed into 68K assembly code.

Within the outermost loop you need to know exactly how many pixels are in each run. That value is stored in the variable `pixelsInRun`, which starts its life at zero. After that you start looking for a run of pixels in the mask.

A run is found by counting the number of nonzero pixels in the scan line. When a zero pixel is found the run has terminated. For every pixel in the run, the variable `pixelsInLine` is decremented, `pixelsInRun` is incremented and the mask pointer is moved along the scan line.

If a run of pixels is found before a zeroed mask pixel, you then have something to compile and you can start hacking away at the run. If a run was not found before a zeroed mask pixel, the compiling section is skipped and you drop into the loop that scans across

the run of zero pixels until it finds a nonzero pixel or runs out of pixels. While this loop is searching for a nonzero pixel it keeps incrementing the variable `runStart` so that when the loop terminates it will contain the number of pixels from the beginning of the scan line to the start of the run.

The run-finding section is a little confusing at first glance, but if you think of it Ping-Ponging back and forth from finding runs to finding antiruns you'll get it. If not, it'll get clearer when a few examples are run through the code.

When (or if) a run of pixels is found, the size of that run along with the output stream pointer and the run's offset are handed off to `EncodeRun`. All three parameters will be altered by `EncodeRun`, hence the need for passing them by reference. Inside `EncodeRun` is where the actual compiling is performed.

The code listing for `EncodeRun` is about 400 lines, and very repetitive. I'll use the *Reader's Digest* version here. For the unabridged version check your source listings.

```
unsigned char * codePtr = *originalCodePtr;

switch(*pixelsInRun)
{
      case 44:
      default:
            if (*runStart)
            {
            // movem.l $xxxx(A0),D2-D7/A2-A6
                  *(unsigned long*)codePtr = 0x4CE87CFC;
                  *(unsigned short*)codePtr = *runStart;

            // movem.l D2-D7/A2-A6,$xxxx(A1)
                  *(unsigned long*)codePtr = 0x48E97CFC;
                  *(unsigned short*)codePtr = *runStart;

                  codePtr += 4 + 2 + 4 + 2;
            }
            else
            {
            //movem.l (A0),D2-D7/A2-A6
                  *(unsigned long*)codePtr = 0x4CD07CFC;
```

```
            //movem.l D2-D7/A2-A6,(A1)
                  *(unsigned long*)codePtr = 0x48D17CFC;

                  codePtr += 4 + 4;
            }

            *runStart += 44;
            *pixelsInRun -= 44;
      break;
```

Once again a local pointer is used for accessing the output buffer. I meant it when I said I really hate typing those "**"s everywhere. Why doesn't C have a *with* thingamabob like Pascal. After getting an easier-to-type pointer to our output buffer, we enter the main core of the function. The `switch` statement branches on the number of pixels contained in the current run.

Each `case` label specifies the number of bytes/pixels that will be copied over by the generated code, with the largest run being the first or default label: 44 pixels. So for any run 44 bytes or greater this section will generate the assembly code that blits over a 44-byte run of pixels. After the proper assembly code is generated, determined by the value of the run offset, the output buffer pointer is bumped up to point past the amount of code produced. The pointer to the current pixel, `runStart`, is moved forward by the number of bytes that will be blitted, in this case 44. The variable that keeps track of the number of pixels in the run, `pixelsInRun`, is decremented by the same amount. With all three of these variables updated the code returns to the caller, who will keep calling this function until `pixelsInRun` is emptied out. On each subsequent call, `pixelsInRun` will be smaller and a matching `case` label will produce a little more code to handle the smaller run.

```
case 43:
case 42:
case 41:
case 40:
      if (*runStart)
      {
      //movem.l $xxxx(A0),D2-D7/A2-A5
            *(unsigned long*)codePtr = 0x4CE83CFC;
```

```
                *(unsigned short*)codePtr = *runStart;

        //movem.l D2-D7/A2-A5,$xxxx(A1)
                *(unsigned long*)codePtr = 0x48E93CFC;
                *(unsigned short*)codePtr = *runStart;

                codePtr += 4 + 2 + 4 + 2;
        }
        else
        {
        //movem.l (A0),D2-D7/A2-A5
                *(unsigned long*)codePtr = 0x4CD03CFC;

        //movem.l D2-D7/A2-A5,(A1)
                *(unsigned long*)codePtr = 0x48D13CFC;

                codePtr += 4 + 4;
        }

        *runStart += 40;
        *pixelsInRun -= 40;
break;

                                .
                                .
Missing cases 39-12. See the real code listing
                                .
                                .

case 11:
case 10:
case 9:
case 8:
        if (*runStart)
        {
        //movem.l $xxxx(A0),D2-D3
                *(unsigned long*)codePtr = 0x4CE8001C;
                *(unsigned short*)codePtr = *runStart;

        //movem.l D2-D3,$xxxx(A1)
                *(unsigned long*)codePtr = 0x48E9001C;
                *(unsigned short*)codePtr = *runStart;

                codePtr += 4 + 2 + 4 + 2;
        }
        else
```

```
        {
        //movem.l (A0),D2-D3
              *(unsigned long*)codePtr = 0x4CD0001C;

        //movem.l D2-D3,(A1)
              *(unsigned long*)codePtr = 0x48D1001C;

              codePtr += 4 + 4;
        }

        *runStart += 8;
        *pixelsInRun -= 8;
break;
```

The sections that handle runs greater than seven bytes all use the same `movem.l` code with only slight variations for the number of registers used. For smaller runs the code drops down to using `move.l`, `move.w`, and drops down to `move.b` as a last resort.

```
case 7:
case 6:
case 5:
case 4:
        if (*runStart)
        {
        //move.l $xxxx(A0),$xxxx(A1)
              *(unsigned short*)codePtr = 0x2368;
              *(unsigned short*)codePtr = *runStart;
              *(unsigned short*)codePtr = *runStart;

              codePtr += 2 + 2 + 2;
        }
        else
        {
        //move.l (A0),(A1)
              *(unsigned short*)codePtr = 0x2290;
              codePtr += 2;
        }

        *runStart += 4;
        *pixelsInRun -= 4;
break;

case 3:
case 2:
```

```
        if (*runStart)
        {
        //move.w $xxxx(A0),$xxxx(A1)
              *(unsigned short*)codePtr = 0x3368;
              *(unsigned short*)codePtr = *runStart;
              *(unsigned short*)codePtr = *runStart;

              codePtr += 2 + 2 + 2;
        }
        else
        {
        //move.w (A0),(A1)
              *(unsigned short*)codePtr = 0x3290;

              codePtr += 2;
        }
        *runStart += 2;
        *pixelsInRun -= 2;
break;
```

In the actual code there are 13 sections that handle runs greater than or equal to 44, 43-40, 39-36, 35-32, 31-28, 27-24, 23-20, 19-16, 15-12, 11-8, 7-4, 3-2, and 1. Each run section after 44 simply drops one of the registers used in the `movem.l`, reducing the number of bytes blitted by four.

```
        case 1:
              if (*runStart)
              {
              //move.b $xxxx(A0),$xxxx(A1)
                    *(unsigned short*)codePtr = 0x1368;
                    *(unsigned short*)codePtr = *runStart;
                    *(unsigned short*)codePtr = *runStart;

                    codePtr += 2 + 2 + 2;
              }
              else
              {
              //move.b (A0),(A1)
                    *(unsigned short*)codePtr = 0x1290;

                    codePtr += 2;
              }
```

```
            *runStart++;
            *pixelsInRun--;

      break;

} // switch(*pixelsInRun)

// Pass back the updated pointer
*originalCodePtr = codePtr;
```

Using the actual code listings let's walk through an example run. If you already understand the flow of the compiler at this point feel free to skip the section.

The first example run will be 107 pixels starting from the beginning of a 128-pixel scan line. After `EncodeScanline` finds our pixel run it passes `EncodeRun` output pointer, the number of pixels between the start of the scan line and the start of the run (in this case zero) and the pixel size of this run (107).

As this run is greater than 44 bytes the `switch` will bop right down to the `default` label. After a quick check of `runStart`, to see if the generated assembly code needs to deal with offset (it doesn't this time), this section generates a 44-byte blit.

```
movem.l   (A0),D2-D7/A2-A6   ;0x4CD0 7CFC
movem.l   D2-D7/A2-A6,(A1)   ;0x48D1 7CFC
```

Following this exciting phase the mundane details of updating the status variables is performed; `codePtr` is bumped up by the size of the written instructions, `runStart` is updated to show that the next section of the current run will start 44 bytes later, and `pixelsInRun` is decremented by the number of bytes removed from this run.

Having chopped off 44 bytes from our initial run size of 107 pixels, the next call `EncodeScanline` makes to `EncodeRun` will be passing a run length of 63 pixels with an offset starting 44 pixels from the beginning of the scan line. Again the run is greater than 44 pixels. Again the `default` branch is taken, but this time the code has to handle the pixel offset. Taking into account the 44-pixel offset, 2C in hex, another 44-pixel blit is produced.

```
movem.l   $2C(A0),D2-D7/A2-A6 ;0x4CE8 7CFC 002C
movem.l   D2-D7/A2-A6,$2C(A1) ;0x48E9 7CFC 002C
```

The run offset is incremented once again, its value now being 88. The output pointer is moved along by the size of the instructions generated; this time you have to take the size of the indirect moves, which are two bytes longer than their direct versions. And the size of the pixel run is again knocked down by 44.

On the next visit to `EncodeRun` the run is 19 pixels with an 88-pixel offset. Finally the run has been reduced below 44 pixels, so you can see another part of the code in action. A run of this size drops us into the 19-16 `case` section, which will produce the instructions to blit over 16 bytes. The same code is generated as before, but this time with about half as many registers. And again the code has to produce indirect moves to account for the 88 (0x58) byte offset.

```
MOVEM.L  $58(A0),D2-D5      ;4CE8 003C 0058
MOVEM.L  D2-D5,$58(A1)      ;48E9 003C 0058
```

After updating all the status variables the code returns what is left of the run back to `EncodeScanline`.

That previous run through `EncodeRun` pared the run from 19 to a measly 3 pixels. And with that massive reduction the code will stop using the `movem` variations. For this subrun the function will produce code to move a single short word at an offset of 88 + 16, or 104 (0x68) bytes.

```
MOVE.W   $68(A0),$68(A1)    ;3368 0068 0068
```

After handling that short `EncodeScanline` is left with a single pixel to pass to `EncodeRun`. For this one pixel with an offset of 106 (0x6A) the `switch` statement will generate a single indirect `move.b`. Hardly worth the effort.

```
MOVE.B   $6A(A0),$6A(A1)    ;1368 006A 006A
```

There. That 107-byte pixel run has been compiled to eight lines of assembly. Not bad. With the masking blitter you would have had to execute 128 logical masking operations for the entire scan line. If you count each logical pixel-masking operation as four lines of assembly, you end up with 512 lines of code to accomplish the same thing as these eight. So the code is shorter, but is it faster? Those `movem.l` aren't free. They burn up 40 CPU cycles per `movem.l` with the number of cycles increasing with the number of registers involved. While the combination of and-ing/ or-ing/ negating to logically mask the pixels chews up about 60 cycles per pixel, both of these cycles' counts are rough estimates due to cache hits, pipe lining, and cosmic rays. Even with a cache fudge factor thrown in the compiling wins hands down in the speed race with logical masking. As with any comparison, your mileage may vary.

```
movem.l   (A0),D2-D7/A2-A6      ;0x4CD0 7CFC
movem.l   D2-D7/A2-A6,(A1)      ;0x48D1 7CFC

movem.l   $2C(A0),D2-D7/A2-A6   ;0x4CE8 7CFC 002C
movem.l   D2-D7/A2-A6,$2C(A1)   ;0x48E9 7CFC 002C

movem.l   $58(A0),D2-D5          ;4CE8 003C 0058
movem.l   D2-D5,$58(A1)          ;48E9 003C 0058

move.w    $68(A0),$68(A1)        ;3368 0068 0068

move.b    $6A(A0),$6A(A1)        ;1368 006A 006A
```

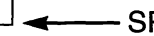
Now that the guts of the compiler have been laid bare it is time to move on and write out the function epilogue.

Compiled Finish

Cleaning up after the compiler is easy enough: write out the function epilogue and then tighten up the buffer that holds the generated code. After those two steps your compiler is done and can return a freshly compiled mask to whoever called it.

At the end of the function the stack holds the contents of the nonscratch registers, return address, and the passed parameters (Table 5-2).

Table 5-2. What the Stack Holds Now
dstPtr
srcPtr
dstPitch
srcPitch
Return Address
Registers d3-d7 & a2-a6



Again using `movem.l` as it is supposed to be used will restore the stashed registers. After restoring the registers a simple `RTS` finishes off the mask compiling.

```
movem.l (a7)+,d3-d7/a2-a6    ; 48E7 1F3E
rts                          ; 4E75
```

You don't have to worry about cleaning up the parameters on the stack; our compiled function is following C calling conventions. Which make the calling function responsible for removing whatever parameters it stuck on the stack.

The code for generating the epilogue is a direct mirror of the prologue code—write out the assembly instructions and move the code pointer along by the size of those instructions.

```
// After compiling write out the function's epilogue
*(unsigned long *)codePtr = kRestoreRegisters
((unsigned long *)codePtr)++;

*(unsigned short *)codePtr = kRTS;
((unsigned short *)codePtr)++;
```

Before returning the compiled mask to the caller you need to resize the output handle to its correct size. This size is found by sub-

tracting the address of the code pointer from the starting point of the buffer. The starting point is the master pointer of the handle so you need to dereference the handle once before subtracting the addresses.

```
// Done writing code, handle may be freed
HUnlock(compiledMaskH);

//Shorten up the code handle
SetHandleSize(compiledMaskH, codePtr - compiledMaskH);
err = MemError();
```

Since even paranoids have enemies, the code checks `MemError` to see if there were any problems in shortening the handle. About the only problem that can occur would be caused by a trashed heap, which would be easy enough to do with a simple error in the compiler. Keep that in mind if you alter this code and start getting memory errors returned.

Using a Compiled Sprite

There are two stages to using compiled sprites: development time and run time. Just as during the development stage you need to recompile your code for every change made, you'll need to do the same for every change made to your game's sprite images. At run time you'll need a way to associate each sprite image with its compiled mask. From this association you can load the mask into memory and execute the compiled code.

My simple approach for the first problem is a little utility program that goes through the resource fork of my game and compiles each color icon it finds and places the output data into a resource of type '68ic' using the same id for this resource as the 'cicn' it compiled. The only problem with this process is that I have to remember to do it. Luckily, altering your game's sprites is performed much less frequently than changing the code. If you are an MPW aficionado you could easily make an MPW tool that compiles the sprites and add this stage to your make file. I'd rather have to remember this stage than be forced to use MPW, but that's me. If you

have a torrid passion for command lines and option key characters, by all means use MPW. Don't think that I don't like MPW. I do. In a weird kind of way it is a love–hate relationship. Love the About Box, hate the scripting language. Any scripting system that requires the use of Key Caps has a fundamental design flaw. But then again it does have one of the coolest About Boxes going.

Instead of precompiling the sprites you might decide that the game could compile them at run time. While your user is gazing at your colorful splash screen you could just load in those icons and start compiling away. This would work but would probably be unbearably slow and error-prone. That it would be slow is obvious; the errors can occur because you will be compiling code into data space and then executing that freshly compiled code. Unless you're careful to flush the processor's caches you can end up executing garbage. Another nit can be future memory protection—if you try to execute code from data space in a future protected OS you'll generate an access violation.

Assuming that the sprites have been compiled into '68ic' resources at compile time, the code for loading and blitting a sprite is straightforward. Load the resource. Lock it down, dereference the handle to get to the function pointer, and then call it as you would with any other C function pointer.

```
typedef void (*CompiledBlitterPtr) (
            long srcPitch,      // bytes to next row
            long destPitch,     // ditto
            Ptr srcPtr,         // Sprites image data
            Ptr destPtr);       // Dest. pixel map

typedef CompiledBlitterPtr *CompiledBlitterHnd;

CompiledBlitterHnd spriteCode;

spriteCode = (CompiledBlitterHnd)

Get1Resource(128, '68ic'); if(   spriteCode )
{
      HLock((Handle) spriteCode);
      (*spriteCode) (srcRowBytes, dstRowBytes,
                           srcPixels, dstPixels);
      HUnlock((Handle) spriteCode);
```

```
        ReleaseResource(Handle) spriteCode);
}
```

In your game, and as shown in future examples, you would not want to load and reload the compiled mask every time you blitted the sprite. You would be wasting so much time you may as well use `CopyBits`. You'd want to load the masks at the beginning of the program or at the start of each level. Which is exactly what later examples will do.

But Which One?

When you start programming your own games you'll need to decide how you want to blit your sprites. Your game might not need to use anything faster than `CopyBits`. Or you might need more speed but find yourself stuck with tight memory limitations. In this case you'll have to compromise between speed and size. All of the blitters presented trade some amount of memory for gains in speed. And then there is that one last messy detail, clipping, to deal with.

Speed

Timing the various blitters in operation is a tricky business. You'll need to decide the average platform you'll support and test your blitters on that piece of hardware. This doesn't mean that you need to run on the lowest-level Mac produced. You just have to run on the lowest-level Mac that you want to run on. If your game only runs well on a 40 Mhz '040 and you don't want to optimize the code to run on a 16 Mhz '020, don't. If your game is not a commercial endeavor then don't sweat it. Then again, if it is you'll have to decide what size of market you're willing to give up. The point is to have fun. Make sure it is clear to your players what your hardware requirements are and be done with it. Players with enough hardware will enjoy your game and those that don't will have an incentive to upgrade. And hopefully with the time you save you can implement a better game, not just a faster one.

As for picking the right blitter, you might want to delay that decision for the player or at least the user player's hardware. Even if you have every sprite compiled, nothing's stopping you from deciding to use a different blitter at run time. Some of the more sophisticated game programmers time their various blitters when the game first starts up and then use the winner for the rest of the game. This is a great idea. If the platform's hardware isn't fast enough to use `CopyBits`, you go ahead and use your own blitters. But if your timings show that `CopyBits` is outperforming your blitter, which could easily happen with hardware QuickDraw accelerator cards, then you get a better blitter and probably a better shot at future compatibility. Other programmers implement the same concept but allow the user to pick which blitters are used through a preference item. Usually the blitter preference is named something like "QuickDraw Compatibility." No matter how you implement it, it's a good idea and one you should keep in mind while designing your game.

Memory

Sprites chew up memory at a voracious rate, and that rate is in direct relationship to their blitting implementation. You need to keep this in mind. No point in having the fastest blitter in the world if it has to use so much memory that no one could afford to play the game. And, like blitting speed, each game will have different memory requirements.

Of the three blitters presented here, each uses the same amount of memory to hold the sprite's source image; height times width is equal to the number of bytes for the source image. The differences lie in the encoding of the masks. Storing the mask as a region usually gives `CopyBits` the smallest mask of the three. But a rare degenerative mask could cause the region to be larger than an equivalent logical blitter's mask. The logical blitter's mask is fixed in size, one byte (due to the expansion) for each byte in the source image. Like `CopyBits'` regions, the mask compiler's memory usage varies with the complexity of the mask. A sprite with lots of holes will produce a mask that is larger than the same size sprite without any holes. If you

look back at the compiler code you can see that a run of one pixel produces mask code of a minimum of two bytes (more likely six). Shove the wrong mask into the compiler and you'll get handed back one huge chunk of mask code. A mask that produces a huge mask will also blit slower than a memory-efficient mask. You get a double bonus for developing sprites with tiny masks, small memory footprint, and faster blitting times. Think about that when you design your next game around a Swiss cheese theme.

Every sprite used within your game increases the game's memory usage (duh!). You'll want to look for ways to reduce the memory baggage associated with your sprites. Besides using a smaller mask you can take advantage of the fact that your game probably uses only a few unique sprites. If you have a few dozen identical-looking baddies on screen at one time, you don't want to duplicate sprite images sitting in memory for each. Write your game so that you can share common graphic images for identical sprites.

The last tip on memory usage is to take advantage of levels. Levels don't just provide a ladder of success for the player, they give you an opportunity to throw away sprites not needed for the next level and to bring into memory even more diabolical evildoers. Using levels allows you to have graphic variations without having to load up all those sprites into memory at program boot time. Plan out not only what you'll load into memory, and what you'll purge, but when. You don't want to try loading sprites in and out of memory while your players are in the middle of a heated battle. Nothing ruins a good game like an inappropriate disk hit.

Clipping

As hard as it is to believe, the previous sprite blitters ignored one important detail: clipping. I'm sure you've experienced clipping in your past programming experience on the Mac. Of course you might not have thought about it. Why should you? When you create a window and then ask QuickDraw to fill a rectangle the size of Montana with a nice vermilion, does it? No. Only the content region of the window is filled with your favorite color, and the state of Montana is spared your attempts at exterior decorating. Quick-

Draw has clipped your graphic request. Anytime you try to color outside of the lines QuickDraw will intervene and force your request to light up only the pixels that lie within your window and the window's port's clipping region. Without QuickDraw clipping your stray pixels you could end up writing over memory that doesn't belong to you. An invitation to disaster. Clipping comes into the picture when the sprite's bounds isn't completely within the destination offscreen (Figure 5-7).

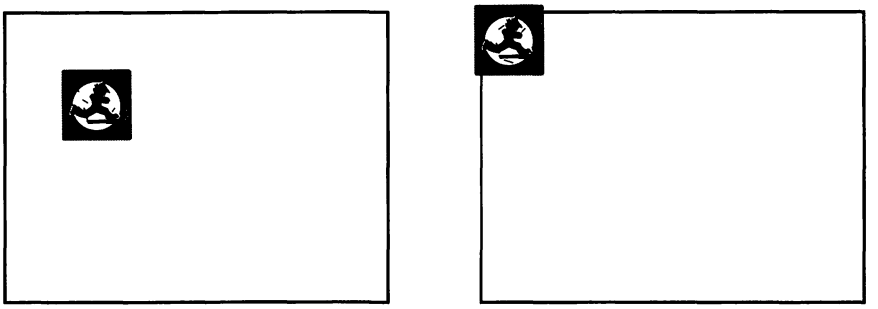


Figure 5-7. Sprite not needing clipping Sprite in need of clipping

Our blitters as written go straight ahead and start blasting pixels without any consideration of where they lie with respect to the destination's offscreens bounds. No problem if the sprite is totally enclosed by the offscreen's bounds; disaster otherwise.

Obviously you need to take clipping into consideration. The easiest way is to use `CopyBits`. Being part of QuickDraw, `CopyBits` will give you the same level of clipping as any other QuickDraw command. If your game requires more speed than QuickDraw can provide, then you have to worry about doing your own clipping.

One quick and dirty way to provide clipping is to design your game so that a situation that would require you to clip never occurs. This is pretty easy to do and for the right type of game would be a satisfactory solution. If you take this route I would suggest adding some debugging code that verifies that sprites are fully enclosed by the destination offscreen. It's easy to slip in a

mistake that would have your sprite slip over the destination's border. And the consequences of that penetration might not be experienced until the chunk of memory that was stomped on inadvertently is used. This could be a microsecond after the said stomping or it could be hours. A bug with hours between cause and effect can be the only incentive you need to start writing accounting software.

If you can't cheat—I mean design—your way out of clipping, you will need to code clipping into your sprite engine. The naive way to do clipping is to look at each pixel that you want to write and after verifying that it lies within the destination's bounds you go ahead and write the pixel into the destination offscreen. This will work. It'll be glacially slow, but it will work. The best way is to follow the example of computer graphics greats and do your clipping on the logical level.

When you ask a graphics package to draw a line that needs to be clipped, it starts off by first asking itself if the line needs to be clipped. Yes, in this example case. Then it finds the intersection points of the line and the legal drawing bounds. If the line does intersect the legal bounds (it might not), the graphic package uses the intersection points as the real line it needs to light up pixels for. The requested line is clipped by finding a smaller line segment for the line that lies within the drawing bounds. Our blitter needs to do the same thing, but only with rectangles.

```
void  LogicalMaskedBlit(PixMapPtr   srcMap,
                        PixMapPtr   dstMap,
                        PixMapPtr   maskMap,
                        Rect *      srcR,
                        Rect *      dstR)
```

The logical blitter presented earlier has enough data passed to it to easily add clipping. First find out if any clipping needs to take place at all. A quick test with the Mac's `SectRect` combined with `EqualRect` will quickly provide the answer and as a bonus will give you the sprite's bounds clipped to the destination's bounds.

```
Rect  clippedRect;

if(SectRect (srcR, dstR, &clippedRect))
{
      if(!EqualRect(&clippedRect, srcR))
      {
      // Prepare the blitter for copying less than the
      // full sprite
      }
      else
      {
      // Sprite fit completely within the destination.
      // No clipping is necessary
            clippedRect = *srcR;
      }
}
else  // Sprite is completely outside the destination
      return;
```

You got the bad news and you do have to clip this particular blit. No problem. Replace the loop controls with the clipped rectangle instead of the sprite's bounding rectangle that it currently is using. Also you need to adjust the destination's starting address to point to the first byte of the destination pixel map after taking into account the clipping. After taking those two steps this blitter is now a clipping blitter. A little slower than before but a lot more versatile.

To avoid the inevitable slowdown imparted by clipping some programmers use two blitters, one with clipping and one without. Others wrap the details in a higher-level function that does the clipping if necessary and then calls a lower-level function that just blits what it's told without asking any embarrassing clipping questions.

Adding clipping to the logical blitter didn't seem too difficult. I wish the same could be said of the mask compiler blitter. Adding clipping to this baby is quite a challenge. The problem is that the compiler has hard-coded all the blitting operations into the code. To account for general clipping you'd basically have to recompile the mask at run time. You don't want to do that. You can hack in vertical clipping in a sprite compiler by taking advantage of the fact that each raster line of the mask is compiled independently of the oth-

ers. After generating the compiled functions prologue, the compiler would need to build a jump table that would index each line of the mask. You'd then write the compiler to execute the blitting code by jumping through the table to only the rows of the sprite that are included by the clipping. This is doable but nasty. And even if you did the necessary retool you'd still have horizontal clipping to deal with. I'd give up and punt at this point. By the time you added general clipping to the compiler you'd have added enough code to obliterate the whole reason for having a compiler.

The solution? Combine the mask compiler with another blitter. You build a general blitter function that determines if clipping is necessary and if not goes right ahead and executes the compiled mask. If clipping is necessary the wrapper function calls another blitter. In my case I call `CopyBits`. I take a big time hit for the clipped blits, but on average most sprites don't need to be clipped. During development I keep a couple counters running. One for each time the mask compiler gets called and the other for when I'm forced to use `CopyBits`. At the end of the games run I drop into the debugger and check out the counters. If the clipping counter was too high I'd redesign the game to avoid this situation as much as possible. I don't know if it's *the* solution, but it's the one I use.

6

Sprite Collisions

What Is a Sprite Collision?

When you fire off your last adamtanium-enriched missile, watching it streak its death arc across the battlefield, ripping into the Vargon mother ship's main lepton reactor and setting off a chain reaction that wipes out the entire Vargon fleet, you have not only saved the world from four-legged creatures with bad haircuts and even worse table manners, you have experienced a sprite collision.

Collisions are the most important part of arcade games, after blitting, that is. Without collisions you'd just have a bunch of sprites running around the screen without any form of interaction.

That's not a game, that's a screen-saver. Collisions are the messengers of interaction. When you get a message that a sprite collision has occurred, you then get to decide what should happen as a result of that collision. This is where your game ideas start to unfold. Sprite A has collided with sprite B. What should happen? Your game. Your choice. I suggest blowing something up.

What Is a Collision?

Two sprites collide when one sprite occupies the same screen real estate as another sprite. In the overdramatized example above, your missile sprite traveled across the screen until it was at the same location as the mother ship. The game detected that these two sprites were occupying the same place at the same time and informed the program. The program kicked in and blew something up.

As you can see, the term *collision* is a misnomer. Sprites don't collide. Collisions require mass and acceleration and momentum and a whole bunch of other physical properties that sprites don't have. Sprites can't collide. They touch. They lie on top of one another. But sprite touching sounded too wimpy for the manly game programmers of long ago, so you're stuck with the aggressive term sprite collision.

Collisions and Holes

Determining if two sprites collide is very similar to blitting the sprites. If the sprites are simple rectangles you can easily see if they collide by performing a quick intersection test on their bounding rectangles. If the bounding rectangles intersect, then the sprites have collided. The tricky part occurs when the sprites are not simple rectangles.

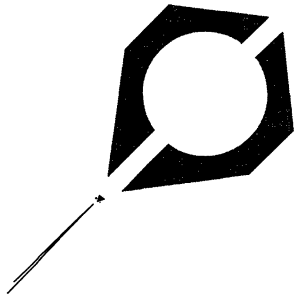


Figure 6-1. Vargon ship with tunnel

If the Vargon ship was constructed with a tunnel down its middle (as in Figure 6-1) and your shot passed cleanly through the tunnel you would not expect a collision to be registered. With only the bounding rectangles to go by, a collision would be generated. If you were playing the part of the Vargon captain you'd be pretty upset that your entire fleet was wiped out by a shot that clearly missed you by a parsec. Your collision detecting needs to account for not only holes but close shots.

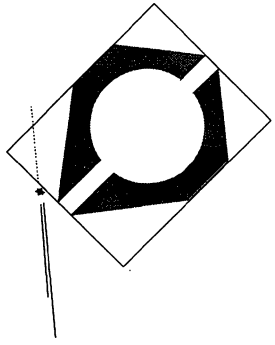


Figure 6-2. Detecting a close shot

In Figure 6-2, the missile will pass by the mother ship without even coming close, yet once again the simple bounding-box method of determining collisions will fail and end up reporting a newly

formed hull breach. And again the player will be outraged that this damn game is obviously cheating.

To correctly determine a sprite collision you need to involve the sprite's masks. Only the mask contains enough information to accurately determine if two sprites have collided. Instead of performing a bounding rectangle intersection test you'll need to perform a mask intersection. If the mask of either sprite intersects with the other's mask, you have a bona fide collision.

CopyBits Collisions

If you're using `CopyBits` with the a mask regions, detecting a collision is as easy as calling `SectRgn`. Pass `SectRgn` your two mask region test candidates and it will give you back a region that equals the intersection of those two regions. Pass this resultant region to `EmptyRgn` and you'll be given a pass–fail grade on the sprites' possible collision. If the region is nonempty your two sprites are involved in a head-on. Empty region? No collision.

```
void SectRgn (RgnHandle maskRgnA, RgnHandle maskRgnB,
                                        RgnHandle dstRgn);
```

You'll have to allocate storage for the result of `SectRgn` with a call to `NewRgn`. I would suggest doing this once and keeping this region handle around as a global. No need to involve the memory manager in a simple collision test.

Logical Masking Collisions

If your blitter choice is the logical blitter you'll want to do your collision testing using the same masks used by your blitter. Checking for a collision turns out to be a heck of lot easier than the masked blitter.

```
Boolean IsLogicalCollision( PixelMapPtr spriteMask1Ptr,
                            PixelMapPtr spriteMask2Ptr,
                            const Rect * sprite1Bnds,
                            const Rect * sprite2Bnds,
                            const Rect * sectBnds)
{
      short       xDiff, yDiff;
      char *      mask1Ptr, mask2Ptr;
      Rect *      bnds1, bnds2;
      short       mask1RowBytes, mask2rowBytes;
      short       xCnt, yCnt;
      short       offset1, offset2;

      xDiff = sprite1Bnds->left - sprite2Bnds->left;

      // verify that sprite one is to the left of sprite 2
      // If not swap our local references so that the mask1
      // is to the left
      if(xDiff > 0)
      {
            bnds1 = sprite2Bnds;
            bnds2 = sprite1Bnds;
            mask1Ptr = spriteMask2Ptr->baseAddr + xDiff;
            mask2Ptr = spriteMask1Ptr->baseAddr;
            mask1RowBytes =
                        spriteMask2Ptr->rowBytes & 0x7FFF;
            mask2RowBytes =
                        spriteMask1Ptr->rowBytes & 0x7FFF;
            offset1 = 0;
            offset2 = xDiff;
      }
      else
      {
            bnds1 = sprite1Bnds;
            bnds2 = sprite2Bnds;
            mask1Ptr = spriteMask1Ptr->baseAddr - xDiff;
            mask2Ptr = spriteMask2Ptr->baseAddr;
            mask1RowBytes =
                        spriteMask1Ptr->rowBytes & 0x7FFF;
            mask2RowBytes =
                        spriteMask2Ptr->rowBytes & 0x7FFF;
```

```
            offset1 = -xDiff;
            offset2 = 0;
      }

      yDiff = bnds1->top - bnds2->top;

      if(yDiff < 0)
            mask1Ptr += mask1RowBytes * (-yDiff);
      else
            mask1Ptr += mask1RowBytes * yDiff;

      for( yCnt = sectBnds->top;
                  yCnt <= sectBnds->bottom; yCnt++)
      {
            for(xCnt = sectBnds->left;
                  xCnt <= sectBnds->right; xCnt++)
            {
                  if(*mask1Ptr++ && *mask2Ptr)
                        return TRUE;
            }

            // move the mask pointers to the next row
            mask1Ptr += mask1RowBytes - offset1;
            mask2Ptr += mask2RowBytes - offset1;
      }

      return FALSE;
}
```

You first find the intersection rectangle of the two sprites. Calculate the two pointers that index into the masks so that both that pointers correspond to the top-left coordinate of the intersection rectangle. Using two nesting loops, work your way across and then down the common rows of the two masks, testing each entry in the masks. If you find a spot where both masks have nonzero entries, you have found a mask intersection. The two sprites have collided. Return that information to the caller. If you make it through both loops you will have exhausted all the common entries between the two masks without finding a collision. Return to the caller your failure to find a collision.

Compiled Mask Collisions

I wish I knew of a way to check for a collision between two compiled masks. I don't. My cop-out solution is the same one as for clipping. Carry around the mask regions that match the compiled masks and use those to check for a collisions. I guess you could write a mask collision compiler. *You* being the operative word here. I'll stick with my cheesy cop-out.

Speed

As you probably guessed, all of these accurate methods of detecting collisions are slightly slower than the simple bounding-box tests. With "slightly" being on the order of a few magnitudes slower. Why do you care about the speed of the collision? Before each frame of game animation your game will want to do all of its collision testing. The longer the collision testing takes, the less time you have for moving your sprites around. No point in having hand-crafted blitters if they spend most of their time waiting for the collision code.

Your best bet for faster collision testing is to avoid the expensive mask-based tests until you have at least checked that the bounding rectangles for the sprites are already intersecting. The bounding rectangle test isn't free, it's just insignificant in comparison to the time consumed by the mask-based tests. And given that most collision tests return false, you don't want to spend any more time on them than absolutely necessary. The rest of this chapter is dedicated to finding even niftier ways to avoid these expensive tests.

What's the Problem?

The main problem with collision testing is not that accurate testing takes too much time, even though it does. The main problem is that time taken for collision testing grows at a nonlinear rate.

When you blit a sprite it takes a fairly fixed amount of time. Add another sprite roughly the same size and shape and your animation phase now takes a predictable amount longer. The growth in the time to blit all the sprites grows in a linear fashion. Linear is good. Linear is fast. Collision detection is not linear.

Picture a game called Solitaire Pong or better yet check out Figure 6-3. In this, the most boring game ever invented, there are only two sprites; the player's paddle and the ball.

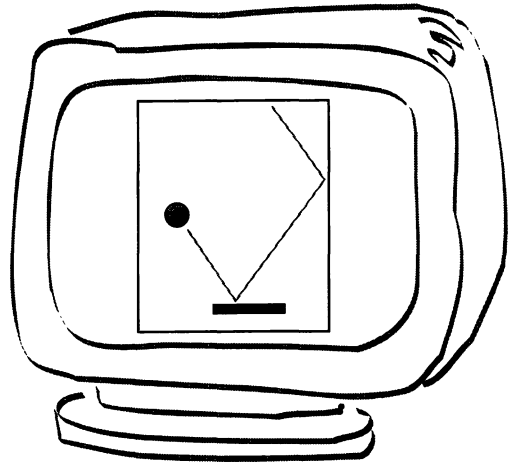


Figure 6-3. Solitaire Pong

With only two sprites on-screen there are only two possible sprite collisions: ball hits paddle, paddle hits ball. Since collisions are commutative, you know if A = B then B = A, so with two sprites you only have to perform one collision test. With two sprites the time eaten up by the collision testing isn't even worth measuring. But what if you add five more balls to the game. Besides creating a more challenging game you have increased the collision complexity by a whole bunch. But how much is a whole bunch? This sounds suspiciously like one of those problems you get in Comp. Sci. 201: Algorithms.

Let's see, given x sprites, to test every sprite for a collision with every other would be x^2. Of course, x^2 means you are testing each sprite against every other sprite including the sprite itself. While the

existential question of whether a sprite can truly ever intersect itself is interesting, for our purpose we're going to go ahead and say yes. Removing self-intersections allows you to remove x sprites from the testing, giving you the equation $x^2 - x$. Remember, if sprite x_1 collides with sprite x_2 you don't want to bother wasting time testing x_2 against x_1. Using this tidbit you can divide the total equation in half, leaving the simple equation shown in Figure 6-4 and leaving me with an excuse to use my equation editor. Maybe later I can work in some excuse to throw in some integrals. I love reading books with integrals. It allows me to justify all that money I spent for college.

$$\frac{(x^2 - x)}{2}$$

Figure 6-4. Collision equation

Now that you have derived the infamous collision equation, you are set to answer the original question. How many collision tests have to be performed with five balls? With the paddle that would make six sprites. A few clicks of a handy HP and you get 15 as your answer. Fifteen tests for six sprites. I hope those tests don't take too long.

A quick look at Figure 6-5, and its corresponding table shows that adding even a few sprites to our game will increase the number of tests performed by some obscene amounts.

Sprites	*# of Tests*
2	1
4	6
8	28
10	45
20	190
30	435
40	780
50	1225

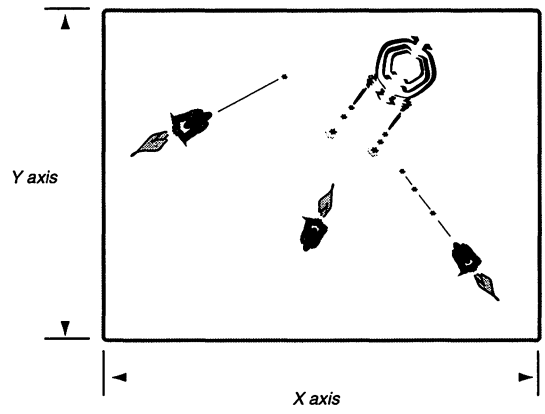


Figure 6-5. Sprite tests

For you algorithmic studs out there the big O notation for this equation would be just plain $O(x^2)$. For the others among us that missed those first few days of Comp. Sci. 201, big O notation is not a way of keeping track of your orgasms. If you need an equation for that, you are wasting your talents programming. Big O notation is the accepted manner of classifying an algorithm's run time performance. In our case the notation $O(x^2)$ means the collision testing run time performance demands increase approximately like x^2. With the big O you ignore the $-x$ and division by two on the basis that their impact on performance is insignificant in comparison to the rate that x^2 grows. It's like ignoring your weight when calculating the weight of the planet; yes, you add some weight but not enough for anyone to care.

Speeding Up Collisions

The solution to speeding up your collision testing lies in rethinking what the problem really is. Instead of thinking of the problem as one of finding the other sprites that are colliding with the test sprite, rephrase it as, Given a sprite search through all of the other sprites, looking for any that share the same screen space. Now sprite collisions has become a classical computer science searching problem. And with classical problems there are classical solutions. Time to dig out one of your dozens of algorithms textbooks. I suggest Sedgewick's *Algorithms*. The fact that it was published by the publisher of this book is purely coincidence. Be careful if you have the copy that uses C for the implementation language. It has a slew of "off by one" errors, with most of them appearing in the sorting chapters. Thought you might want to be warned, as that is the optimization I'll use first.

Sorting

One way to speed up a searching problem is to place the data to be searched into a well-known order. If you were searching for a zip code among thousands, a good approach would be to sort the zip

codes into ascending order before searching. With the zip codes sorted you could then apply a more sophisticated searching technique like a binary search to find the zip code you are looking for.

The same idea can be used for sprite collisions. After you move each sprite to its next location for the next frame and before you start the blitting for that frame, sort the sprites according to their new on-screen locations. Then for each sprite you can quickly look to see if any other sprites overlap. If you do find a sprite that overlaps you then only have to check the few sprites surrounding the first for additional collisions. You stop checking the surrounding sprites when you find a sprite that is over half the distance of the test sprite away from the original hit sprite. Because of the sorting you're guaranteed that no other sprites can overlap once you find one that doesn't. After checking both sides of the sorted list around the initial hit sprite your collision testing is done. And you will only have had to test a small sample of the sprites on the screen.

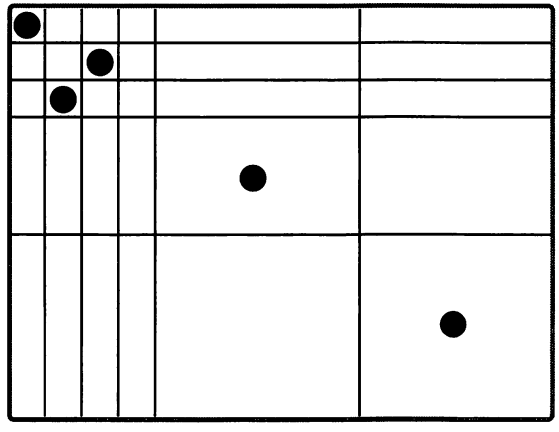


Figure 6-6. Sorting sprites

You'll have to decide on what axis, x or y, you want to sort your sprites. You want to pick the axis that will give you the most even distribution of sprites across the screen. In the oversimplified Figure 6-6 you'd want to sort on the x axis. The x axis usually is the predominant candidate for most games, as there are usually more x

pixels than y pixels. More pixels means more space for the sprites, although the axis you pick will be eventually be driven by your game's layout.

After deciding on what axis to sort the sprites you'll need to decide how to sort them. In choosing your sorting algorithm keep in mind that each of the sprites moves only a few pixels at a time. You'll want to choose a sort that deals well with already sorted data, as that is what you'll have most of the time. Insertion sorts and radix sorts work well within these constraints. Heck, even the much-maligned bubble sort could be used here without fear of ridicule. Or you might forgo a sorting algorithm and just move each sprite within the sorted list after you calculate its new position. The sprite will usually only have to move up or down one place in the list. That is if it has to move at all. Most of the time it won't.

No matter how efficiently you sort your sprites you are burning up some amount of processor cycles. You only want to resort to sorting if the costs of sorting are covered by the savings in searching. In other words, don't jump right into sorting your sprites until you've seen whether your game can get by with the simple brute force methods of collision detection.

Sectors

Another way to speed up collision testing is to apply the old divide-and-conquer axiom. Instead of testing for collisions on a huge bunch of sprites across only one screen, divide the screen into several smaller screens with each divided screen holding only the sprites that lie within that screen's bounds. Let's call each of these divided screens a sector. Your collision testing procedure becomes one of first finding the sector your test sprite is currently lying within. After that you need only test that one sprite against the other sprites that are also in the same sector. You still have a $O(x^2)$ performance bounds, but you have reduced x to a reasonable number.

If you decide to try the sector method you want to make your sector sizes such that they are a power of two larger than the

matching dimension of the largest sprite. If your largest sprite is 30 ˘ 56, then you want to your sectors to be 32 ˘ 64, the closest powers of two that are larger than the sprite's dimensions. With the sector size based on powers of two you just have to shift your sprite's coordinates to the left to find what sector the sprite belongs in. Much better than a nasty divide.

Sectoring the screen sounds easy enough, but, of course, there is a catch. The catch is what to do about sprites that lie on sector boundaries. A sprite could easily lie in one, two, or four (and eight if you allow objects bigger than sectors) sectors at once. If this situation were to occur you would have to check each sector the sprite intersects for possible collisions. This isn't all that slow as it is annoying.

When I was writing this section I stumbled across a posting that shows that shifting the sector grid four times gives better performance than a static sector grid and then checking for overlaps.

This careful analysis is brought to you by Tom Moertel. You got to just love any analysis that uses the word *asymptotically*.

```
ANALYSIS: FOUR-SHIFTS vs. ADJACENT-SECTORS

Before you begin thinking that this shift-and-
repeat technique is terribly inefficient, consider
the alternative, checking adjacent sectors. Let's
say you've got a sector in the middle of the screen;
call it S. Objects in S could collide with objects in
adjacent sectors, so you'd have to include all eight
of them in your collision testing of S. How does that
affect running time?

Assume that objects are randomly distributed over
the screen and that there are on average K objects in
each sector. Recall that to test for collisions in
each sector, we use a brute force technique that re-
quires n(n-1)/2 rectangle intersection operations
(check it) for n objects. Now we can compare the
four-shifts method with the test-adjacent-sectors
method.

. . . Four-shifts method: each sector is checked
by itself, at a cost of K(K-1)/2 rectangle tests, but
```

the process is repeated 4 times. Consequently, the cost to entirely check a sector is 4 * K(K-1)/2 = 2K(K-1) = 2K^2 - 2.

. . . Adjacent-sectors method: Each sector is checked only once, but its eight neighboring sectors are included in the check. Define L = (1+8)K be the average number of objects in these 9 sectors. So the cost per sector is L(L-1)/2 = (9K)((9K)-1)/2 = (81K^2 - 9K)/2.

Now, let's calculate the ratio of the two methods' expected number of rectangle tests:

$$R = \frac{\text{cost of adjacent-sectors}}{\text{cost of four-shifts}} = \frac{(81K^2 - 9K)/2}{2K^2 - 2}$$

Note that the limit of R as K -> Infinity is 20.25. Asymptotically, then, the four-shifts method is about 20 times faster than the adjacent-sectors method. Admittedly, it's unlikely you'll have an infinite number of objects on the screen. That fact begs the question, how much faster is the four-shifts method for the more common cases in which there are, on average, one, two, or three objects in a sector? Answer: For one object, it's *much* faster; for two, 38 ˜ faster; for three, 30 ˜ faster.

The four-shifts method needs to perform *no* tests when there's only a single object in a sector—a very common case. The adjacent-sectors method, on the other hand, needs an average of 36 tests to handle the same situation.

Cool. Thanks again Tom.

Dividing up the screen will only give you better performance if your sprites are evenly distributed across the sectors. A game that has hundreds of swarming bees surrounding the player, forcing all those sprites into the same sector, will end up with $O(x^2)$ perfor-

mance. If you find yourself in this situation, writing a game with bad clustering not surrounded by hundreds of swarming bees, you can take a hint from ray tracing algorithms.

Ray tracing algorithms have some of the same problems of collision detection. As each ray is shot into the modeled scene the programmer must find the objects that the ray intersects with as it passes through the 3D world. Since the programmer will be shooting a gazillion rays into the scene, she needs a way to efficiently find what object will intersect the rays. Sound familiar? This is a heavy area of research and one of the cooler ways found is to partition the 3D world in the same manner as the sector collisions. The main difference is that the 3D rendering folks divide their space around the objects themselves, effectively creating a hierarchy of sectors in which the more objects there are, the more sectors there are. If you applied the same idea to collision detection you would have a sectoring scheme that did not require the sprites to be evenly distributed across the screen.

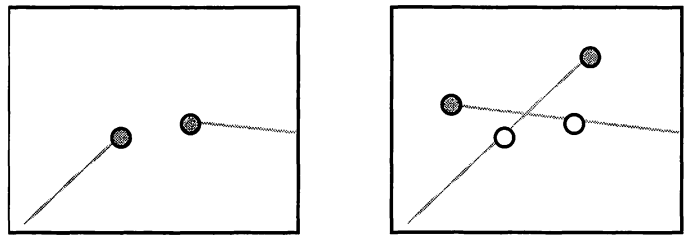


Figure 6-7. Sector-mapping the screen

Simplifying by Design

Now that you've explored the more advanced methods of collision detection let's take a look at the simplest way to reduce collision testing time: design around the problem. All of the previous examples of collisions assumed that each sprite has to test against every other sprite for a possible collision. This would be true only if your

game's design needed every sprite to collide with every other sprite. Very few games match this description.

In almost every game designed, the sprites can be divided into three categories: player's, enemy's, and maintenance. The player sprite would be the sprite that represents the player and anything the player can launch—missiles, lasers, bombs, bananas. Enemy sprites are the player's targets along with anything they might start slinging at the player. Maintenance sprites are those sprites you don't pay much attention to—your score, floating bonus tags, explosions, miscellaneous debris. With this categorization you can reduce the number of possible collisions by programming within a few constraints. Like: the players can't shoot themselves, the enemies can't shoot themselves, and nobody can shoot the maintenance sprites. With these rules you've reduced the number of possible collisions by at least two-thirds. You don't have to bother checking each player missile with any of the other player missiles or the player. Same goes for the enemies; their missiles pass right through their brethren and only collide with the player.

You'll have to program your game to enforce these rules. In the above example you might keep three lists of sprites, one for each category. When it comes time to start testing for collisions you'd test the player's list against the enemy's list and then repeat the process for the enemy's list. You'd skip the maintenance list altogether.

Let's use the above rules with an example. There is one ship for the player who has fired four missiles. There are 30 enemy ships with 7 missiles currently homing in on the player. And don't forget the maintenance sprites, one for the current score, another for the high score, and one more for the current level. Thirty-eight sprites total, with the brute force number of collision tests at 703. I hope those tests don't take too long. By applying the game's rules you get 37 tests by testing the player's ship against the enemy sprites and their deadly missiles. Each of the player's missiles must be tested against the enemies and their missiles, adding another, let's see 4*30 + 4*7 = 148, 148 additional tests. The maintenance sprites contribute a big fat zero. Giving us 37 + 148 + 0 = 185 tests total. So by programming a few constraints into the game you're able to re-

duce the number of possible collision tests from 703 to the slightly more reasonable figure 185. And you always thought game programmers were just being nice by not letting you blow up your own ship.

Frame Rates and Collision Detection

Collision detection requires that your sprites don't move too much during one frame of animation. By having your sprites take large leaps for each frame you'll end up missing collisions. The problem is temporal. Your frames of animations are snapshots of time. As with a strobe light at a disco, you don't see what happens between the flashes of light. You're only able to see during the flashes. Between two flashes of the strobe you only see the start and finish of a suave disco move. In between the two flashes you only hear the faint rustling of Angel Flight slacks. You miss all the visual movement in between. Each frame of your game provides the same sensation as that disco beacon.

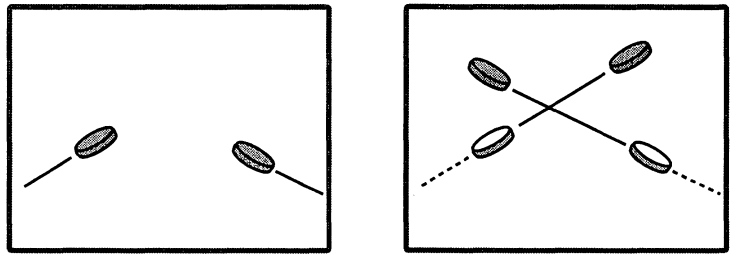


Figure 6-8.

Between any two frames of your game you could potentially end up with the problem illustrated in Figure 6-8. Two, uh, pucks I guess, are moving toward a sure edge-on collision in frame one. Between frame one and frame two the game has moved the pucks too many pixels forward in time, bypassing the preordained collisions.

Your players won't like this. Their brains have already calculated the pucks' trajectories and laid 100–1 odds that they'll collide. Cheat the brain and it will seek revenge by creating an emotional response in the player, "This game sucks!" being the typical gypped brain response.

> If you can't crank your frame rate high enough to provide for pixel-based collision detection, you can always fall back on your high-school physics to find your collisions points.
>
> Instead of moving the sprites and then seeing if they have collided, and maybe missing a collision or twenty, you could calculate your sprites' trajectories as vector equations and algebraically determine if they collide. You'll be removing the sampling of time from the collision process and applying a continuous function to the task. You'll gain mathematical accuracy at the cost of processing time and you might end up with a collision calculating out to occur at frame 3.236 instead of a nice wholesome number like 4, but you won't ever miss one.

To avoid this situation you need to balance the speed of your targets with the speeds of the player's projectiles and the animation frame rate of your game. To insure accurate collision detection your sprites need to move only a few pixels, at most, between each animation frame. But to have fast-moving targets or faster-moving missiles, they need to cover large screen distances in small amounts of time. The equalizer is frame rate. A faster frame rates allows your sprites to make smaller leaps between frames. Smaller leaps provide for more accurate collision detection. And accurate collision detection leads to happy players who'll keep playing your games. Happy players who keep playing will keep buying. Buying players make you more money. More money makes you rich. Rich game programmers stop writing games, allowing a new generation to start programming games. These new players will need an introductory game programming book. Ta-da! Therefore accurate collision detection equals rich author. Isn't predicate logic fun? Next time we'll watch Rush Limbaugh and play Spot the Fallacy!

7

Game Plan

With the previous chapters you constructed the graphical building blocks of knowledge needed for game programming. It's now time to take these Legos of technology and start snapping them together into larger objects. In the next few chapters you'll take this kit bag of knowledge and build a game-programming class library. The first pass of this library will encompass everything you've read—offscreens, sprites, blitter, collision detection, everything. With this library you'll have a foundation to start programming your own games.

Design Goals

This library needs to provide a foundation for the graphical elements needed to program games on the Mac. Easy enough. Its main goal is ease of understanding for the game programmer. This library's purpose on Earth is to help you to learn write your own Mac games. The library will not carry you from your first fumbling game programming steps to the time you decide to hang up your carpal tunnel braces, but it will provide hooks for your own customizations. When you fully understand the limits of this library and are able to sit down and fix them, you'll have snatched the pebble from my hand and earned the right to pave your own way in the game programming world. Don't forget to stop and brand yourself on your way out.

Easy to understand library that allows for customization. Not a bad signature for a library. To accomplish these goals the library will be written in C++. C++ allows the core code to stay in plain old C and still have the advantage of customization through its class inheritance mechanism. To gain customization through inheritance the library will create classes for the major elements of the library.

Other than C++ classes and inheritance, the library will ignore the other interesting features of C++. No operator overloading, no streams, no exceptions. No exceptions. I know recoding the '+' operator to balance your checkbook and change your car's oil instead of simply adding two tiny numbers is fun. But you'll have to do that on your own time, ditto with all the other shiny linguistic features that C++ brings on Christmas morning. Classes and inheritance are all you'll need for the library. If you're lucky enough to be an operator overloading, paramaterized types using, i/o stream streamer, Shao Lin master of C++, then feel free to fold in as many C++ master moves as you want.

C++ and Games

The trouble with using C++ as an implementation language for our library is its reputation. For some reason C++ has the reputation of being slower than C. And game programmers, the speed

junkies that they are, have believed these distasteful rumors and have avoided C++ in droves. Is there any truth to the rumors? Depends. C++ in its full glory is a hefty language. If you use each of the features of C++ without a feel for their costs, you could easily produce code that runs slower than the functionally equivalent C code. This isn't C++'s problem, it's just a tool. Tools need to be used properly.

The root of C++'s tarnished reputation is its inheritance mechanism. Programmers brought up on the concept of no free lunch look at inheritance and see a huge black box that will suck processor cycles faster than M&Ms disappear at a Weight Watchers meeting. Since our library will be grounded in inheritance, now would be good time to see if there is any truth to this nasty rumor.

For those of you who have been stuck in a DP cave writing COBOL applications, inheritance is a mechanism of a programming language that allows the programmer to specialize or completely replace the behavior of a preexisting chunk of code. In C++ this chunk of code must belong to a class and is called a member function. I hope a simple example will make things clearer before I muddy them up with implementation details.

> For you language lawyers out there, I sometimes refer to C++ member functions as methods and, occasionally, member data as instance variables. If this bothers you I'm sorry. I was raised on a steady diet of Object Pascal and it's a hard habit to break. Plus, I break out in a Beavis and Butt-head-like giggle whenever I say "member function."

To inherit from a class you must have a class to inherit from. Most C++ texts refer to this class as the base class. Usually the base class provides an interface that the derived classes can specialize for their own needs. Let's declare a base class of "Car." The C in front of the "Car" is a tag that quickly lets the reader of the code know that the variable being played with is a C++ class. Other conventions use the letter T, for *type*, I think.

```
class CCar {
public:
      virtual float TopSpeed() {return 55;}
};
```

Suppose a simple class with `TopSpeed` as its only method, which will return the top speed the car is capable of in miles per hour. If you instantiated (one of my favorite words) a car of this class you would get 55 as its top speed.

```
class CLotusEsprit : public CCar {
public:
      virtual float TopSpeed() {return 167;}
};
```

A descendant of the base class CCar, CLotusEsprit, has inherited the `TopSpeed` method from CCar and replaced it with code that returns a top speed that your insurance company would be impressed with.

One of the advantages to inheritance is that code that uses the objects doesn't have to know the exact identity of the object that it's playing with. Knowing about the base class is good enough.

```
void PrintTopSpeed(CCar & car)
{
      printf("Top speed: %f\n" car.TopSpeed());
}

/*-------------------------------------------------
      Driver function that creates a few car objects
      and prints their top speed.
-------------------------------------------------*/
void TestDriver()
{
      CCar            simpleCar;
      CLotusEsprit    fasterCar;

      PrintTopSpeed(simpleCar);
      PrintTopSpeed(fasterCar);
}
```

Here the driver function creates two car objects and then passes them to `PrintTopSpeed`, which will request the car to output its top speed. The cool thing is that `PrintTopSpeed` only knows about one type of object, CCar. But through the magic of inheritance it will still write out the correct top speed of the Lotus. This nifty trick is what worries game programmers so much. For C++ to pull off this stunt it has to determine at run time what type of car object is really being asked to show its top speed and then call that object's specific implementation of the `TopSpeed` method. How long it takes C++ to look up the proper method is the problem. So how long does it take? In the immortal words of that Tootsie Pop commercial, "Let's find out!"

To determine the costs of inheritance we'll need a baseline to measure from. In this example let's use CCar without any descendants and without the virtual specifier. An additional method, `TurboCharged`, has been added to help make the assembly dumps clearer.

```
class CCar {
public:
      Boolean       TurboCharged();
      long          TopSpeed();
};
```

If the code creates a CCar object and then calls its `TopSpeed` method, your compiler will produce the same code it would for any other function call.

```
main()
{
      long       scratch;
      CCar       plainCar;

      scratch = plainCar.TopSpeed();
}
```

This tiny driver function, according to Macsbug, will output this simple three lines of assembly. The only tricky C++ thing is the parameter being pushed on the stack before the function call. Even though `TopSpeed` doesn't ask for any arguments in its declaration,

C++ will always pass one to it: the pointer to "this." The "this" pointer provides a pointer to the object and is pushed onto the stack for every method of every class you use in C++. So without any inheritance C++ has already bloated a single function call with an extra line of code.

```
PEA       -$0004(A6)                                    | 486E
JSR       'CODE 0002 23A0'+000C       ; 01D472DC        | 4EBA
MOVE.L    D0,D3                                         | 2600
```

To see how much inheritance adds to the code from the baseline you'll need to redeclare the methods to be virtual and to provide some descendants of the base class CCar.

```
class CCar {
public:
      virtual      Boolean TurboCharged();
      virtual      long TopSpeed();

};

class CLotusEsprit : public CCar {
public:
      virtual long TopSpeed();
};

class CLotusEspritTurbo : public CLotusEsprit {
public:
      virtual Boolean TurboCharged();
      virtual long    TopSpeed();
};

class CLotusEspritTurboS4 : public CLotusEspritTurbo {
public:
      virtual long TopSpeed();
};
```

Four descendant classes will insure that the compiler can't do any optimizations for the method look-up.

```
long PrintTopSpeed(CCar & car)
{
      return car.TopSpeed();
}

main()
{
      long                    scratch;

      CCar                    plainCar;
      CLotusEsprit            plainLotus;
      CLotusEspritTurbo       fasterLotus;
      CLotusEspritTurboS4     thisYearsLotus;

      scratch = PrintTopSpeed(plainCar);
      scratch = PrintTopSpeed(plainLotus);
      scratch = PrintTopSpeed(fasterLotus);
      scratch = PrintTopSpeed(thisYearsLotus);
}
```

Each call to `PrintTopSpeed` will pass a reference to a specific car object, but as far as `PrintTopSpeed` is concerned it only gets handed a reference to a CCar object. This forces the function to use inheritance to find the correct method to call. A quick trip to Macsbug reveals the number of instructions required to implement inheritance in C++.

```
LINK      A6,#$0000
MOVE.L    $0008(A6),-(A7)
MOVEA.L   $0008(A6),A0
MOVEA.L   (A0),A0
MOVEA.L   $0004(A0),A0
JSR       (A0)
UNLK      A6
MOVEA.L   (A7)+,A0
ADDQ.W    #$4,A7
JMP       (A0)
```

Ignoring the function prologues and epilogue you end up with only four lines of code that contribute to the method look and eventual execution.

```
MOVEA.L     $0008(A6),A0
MOVEA.L     (A0),A0
MOVEA.L     $0004(A0),A0
JSR         (A0)
```

Again the first line pushes the "this" pointer onto the stack. After that the code dereferences the object until it gets to the jump table of methods for that object. After loading the address of the correct method the code indirectly jumps to the proper implementation of `TopSpeed`. Usually the jump point will be into your applications jump table, which will require another jump before really getting to the proper method.

Four or five instructions is all it costs for inheritance. Not a bad price to pay for the design elegance of inheritance, but a cost nonetheless. You wouldn't want to have a method look-up for every pixel you process during a blit, but one method look-up that determines the correct blitter to invoke wouldn't be too bad. It would probably be faster than using a nest of if-elses to pick the proper blitter.

However, four or five instructions aren't the only cost of inheritance. That jump table used by the classes takes up a few bytes of memory. And the number of bytes used increases as the number of methods is added to each class. Not a big deal, but like the five additional instructions, something to keep in mind while designing your games.

The Pieces

Based on the previous sections of this book, the main classes that cry out to be coded are, in no particular order of appearance: sprites, of course; a wrapper object for each frame of the sprite's image; and a place for the sprites to play that handles the double buffering and other animation details. And in order to reduce collision detection, a class to group sprites into rule-based units; good guys, bad guys, and maintenance guys.

Given the four classes, you'll need four name tags for them. The class that manages the double buffering will be called a Play field. Sprite I think can stay as is. I like Sprite Cel for the single frames of animation for the sprite. And Sprite Group sounds like a good name for a class that groups sprites.

The four classes will be grouped in a hierarchy formation. Play fields will be the root container class. Play fields will hold all of the sprite groups used by the play field. Each sprite group will hold a list of sprites. Every sprite can hold zero or more sprite cels, as in Figure 7-1.

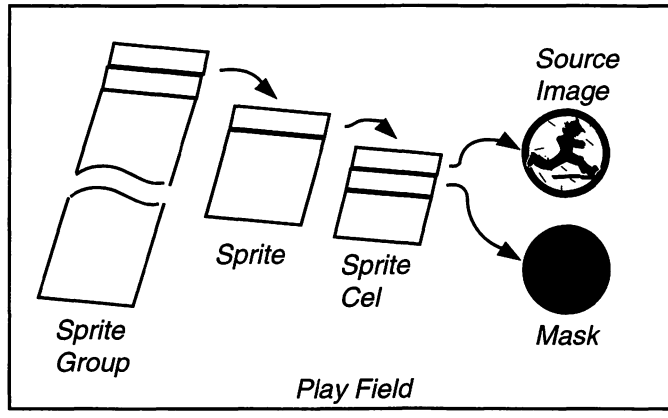


Figure 7-1. Class hierarchy

To implement this hierarchy the classes will need a way to link all the pieces together. So you can add another class to our shopping list, a list class. A shopping list that has a list on it, how LISP-ian.

Lists

Our design calls for a simple list class that will allow the code to store sprite groups, sprites themselves, and sprite frames. The list must provide easy access to each entry on the list, a count of the ob-

jects currently in the list, and a path to iterating across the entries on the list.

For ease of use let's base the design of the list class around the idea of a variable length array. A variable length array is an array that allows you to add or delete entries at run time, unlike C arrays, which are fixed in size at run time.

```
class CObjectList {
public:
      // Creation and Destruction
      CObjectList(long initialItems = 4);
      ~CObjectList();

      // Status
      long        GetObjectCount() { return fCount;}
      Boolean     IsEmpty() { return fCount == 0; }

      // Inserting and removing from list
      void        Add(void * object);
      void        Remove(void * object);

      // Indexing through list
      void *      GetNthObject(long index);

private:
      long **     fArray; // Holds the handle of objects
      long        fCount; // Number of objects in list

      void        ResizeArray(long slots);
      long        ObjectOffset(long index)
                  { return (index-1) * sizeof(void *); }

      void *      ObjectPtr(long index)
                  {
                        return &((char*)*fArray)
                              [ObjectOffset(index) ];
                  }

      long        FindIndex(void * object);
};
```

The implementation of the list class provides no insight into game programming. So let's skip it. If you're curious go ahead and check out the source package supplied. Before heading on to the

real classes let's spend some time going over how you can use the list class.

Creating a list is performed like any other C++ class. You can pass the number of slots you want preallocated to the list's constructors. A default of four has been provided, so you can ignore the details of the class.

The list's destructor throws away the Mac handle that holds the pointers to the objects stored in the list. Out of paranoia the destructor zeroes out the object count and the array storage handle.

```
CObjectList *       list;
CSprite             sprite;

list = new(CObjectList);

list->Add(sprite);
list->Remove(sprite);

(list);
```

Creating, adding, and then removing an object (in this case a sprite object) and then disposing of the list takes only a few lines of code. Taking the list class along with us, let's begin building the sprite class kit, starting with the play fields.

Handling Errors

Regrettably, our class library will have to provide some form of error notification. It's nasty but it has to be done. The class kit will respond to any abnormal condition by calling an error handler passed in by your program at boot time. What you choose to do at this point is up to you. You could simply quit, as just about any error returned by the sprite library is likely to be fatal, or you can use the callback as a spot to throw a C++ exception.

At the beginning of your program, before actually using the sprite library, you'll want to install your error handler. Call `SetSLVGErrorHandler` with your function pointer and any other long value that you'd like to be passed back to your error handler.

```
void SetSLVGErrorHandler(SLVGErrorProcPtr errorProc,
                                                long refcon);
```

If you don't install an error handler you'll be stuck with the default error handler. You don't want this. The default error handler simply makes a `DebugStr` call with a string that shows the OS error that tripped up the library. If you don't have a debugger installed you'll be greeted by a unimplemented trap error. Your error handler will follow this function prototype.

```
void MyErrorHandler(OSErr error, long refcon);
```

If you ever need to retrieve the last error encountered by the sprite library, you can call the error support function `GetLastSLVGError`.

```
OSErr GetLastSLVGError(void);
```

This function will return the last error condition that the library experienced even after the error handler has been executed. The result of this function is "sticky." That is, by calling the function you don't clear the error. Subsequent calls will return the same error code unless the sprite library has performed an action that has either cleared the error or created a new and different error.

8

Play Fields

The play field class will be the central focus of the sprite library. The three main stages of your game—moving sprites, checking for collisions, and, finally, animating the sprites onto your Mac screen—will be the responsibility of the play field objects.

```
CPlayField *gameField;

// Play field and misc. sprite creations skipped

while( GameStillRunning() )
{
      gameField->MoveSprites();
      gameField->CheckForCollisions();
      gameField->ShowNextFrame();
}
```

A game loop is like an event loop but is driven by the stages of sprite animation instead of external events. The break condition for the loop depends on your game. The loop usually breaks when the player runs out of lives or pauses the game. Inside the game loop is where all the fun stuff happens with play fields. All the sprites belonging to the play field are first moved, then checked for potential collisions. The play field is then requested to show the results of the previous two stages through the play field's `ShowNextFrame` method.

Double-Buffered Animation

Since one of the main purposes of the play field is to manage the double buffering animation process, it would be a good idea to explore that process before going on.

The purpose of double-buffering animation is to provide flicker-free animation of sprites over a background image. Double-buffered animation achieves this goal by providing two offscreen buffers and the screen itself to build one frame of animation (see Figure 8-1). Assuming that the animation will appear within a window on the Mac screen, the two offscreen buffers will be the same dimensions as the on-screen window. The first offscreen buffer holds a copy of the background image. The other buffer is a scratch, or workplace, buffer. With the two buffers allocated and the first buffer holding a copy of the background, the animation process works in these four steps.

First, move each sprite to its new location. For each sprite, construct a rectangle that is the union of the sprite's previous location and its new location.

Next, copy the pixels that fall within the combined locations of the sprite from the background buffer to the workplace buffer. This pixel copy will erase the previous location of the sprite with a piece of the background. If you remember, this step was an optimization from copying the entire background for each animation frame. This optimization is referred to as dirty rectangle animation.

The third step is to blit the sprite's current frame to its new location within the workplace buffer. Only the pixels included by the sprite frame's mask are copied over during this step.

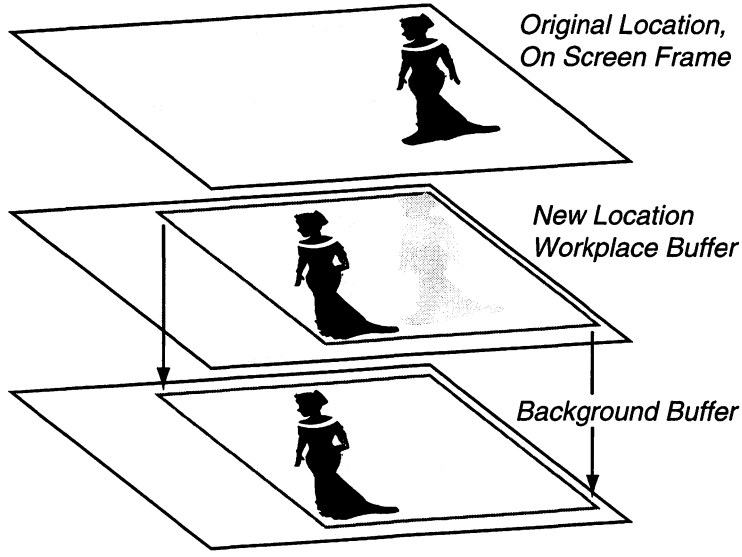


Figure 8-1. Buffering scheme

Finally, copy over all the areas of the workplace buffer that have changed from the previous animation frame to the on-screen window.

The play field will perform the sprite's erasure and the final blitting onto the Mac's screen stages itself. It will only orchestrate the sprite's movements and the sprite's blitting of its own internal frames. The actual work will be performed by the sprites themselves at the request of the play field.

Play Field Creation

To create a play field object, its C++ name being `CPlayField`, you need a window and a big chunk of available memory. The memory will be eaten up by the two offscreen buffers needed by the play field. The constructor also takes a rectangle that describes the area of the window that the sprites should animate across. If you pass a nil pointer for the rectangle pointer the play field will automatically create a play field the size of the window passed in.

```
CPlayField(CWindowPtr * hostWindow, const Rect* hostBounds);
```

Because the library works only on color-capable Macs, the window parameter is expected to be a color window.

The constructor spends most of its time creating the offscreen buffers. Both buffers are implemented using GWorlds, which are created using the play fields utility function `CreateGWorld`.

```
CPlayField::CPlayField(CWindowPtr * hostWindow,
                       const Rect* hostBounds)
{
      OSErr err = noErr;

      fBkgndBuffer = nil;
      fWorkplaceBuffer = nil;
      fHostPort = nil;
      fSpriteGroups = nil;

      ASSERT(hostWindow != nil);
      fHostPort = hostWindow;
      fHostBnds = hostBounds == nil ?
                       hostWindow->portRect : *hostBounds;

      // Create the offscreens
      err = CreateGWorld(&fBkgndBuffer, &fHostBnds);
      if(err)
      {
            fBkgndBuffer = nil
            PostFatalError(err);
            return;
      }

      err = CreateGWorld(&fWorkplaceBuffer, &fHostBnds);
      if(err)
      {
            fWorkplaceBuffer = nil
            PostFatalError(err);
            return;
      }

      // Create the list that will hold all the groups
      fSpriteGroups = new(CObjectList);
      if(!fSpriteGroups)
            PostFatalError(memFullErr);
```

```
        // Create the list that will reference all the sprites
        fAllSprites = new(CObjectList);
        if(!fAllSprites)
                PostFatalError(memFullErr);
}
```

Both buffers are created at the depth of the deepest monitor that the host window intersects. If there was enough memory to create both buffers, the code constructor continues constructing by building an empty object list. This list will eventually hold the sprite groups that reference all the sprites involved. The constructor finishes off by building a list that will hold a reference to every sprite in the play field. This list will save the play field the effort of having to work through each of the groups to get to the individual sprites.

```
CPlayField::~CPlayField()
{
        // Kill off the background buffer
        if(fBkgndBuffer)
                DisposeGWorld(fBkgndBuffer);
        fBkgndBuffer = nil;

        // Ditto with the working buffer
        if(fWorkplaceBuffer)
                DisposeGWorld(fWorkplaceBuffer);
        fWorkplaceBuffer = nil;

        // Get rid of the master sprite list
        if(fAllSprites)
                (fAllSprites);
        fAllSprites = nil;

        // Get rid of all the sprite groups
        if(fSpriteGroups)
        {
                fSpriteGroups->FreeAll();
                (fSpriteGroups);
        }
        fSpriteGroups = nil;
}
```

Disposing of the play field is a simple matter of throwing everything away that was allocated in the constructor, taking care that what is going to be disposed has actually been allocated. An error in constructing could result in the destructor's being called with the play field in a partially constructed state. This is not an unlikely scenario given the size of the offscreen buffers the play field is likely to create.

The sprites contained in the sprite groups are destroyed completely before freeing the group list owned by the play field. By freeing the sprites in the play field destructor you can forget about having to free them individually. One call to free the play field should be enough kill off everything associated with the play field. You free the master list of sprites without having to worry about the sprites it references. They'll all but have disappeared by then.

The only thing you have to be careful about is having two play fields referencing the same sprite. When you dispose of one of play field it will destroy all of the sprites that have been attached to it. No problem yet. The fun happens when you dispose of the other play field. When it reaches the point where it disposes of all of its sprites, it will attempt to call a destructor for a sprite that no longer exists. BOOM! If you're lucky. A trashed heap that you don't discover until a few hours later if you're not.

Adding Sprites to a Play Field

After successfully attaching a play field to an on-screen window you can start adding sprites to that play field. More accurately, you can start adding sprite groups. The process for adding sprites is to first add the sprite to a sprite group and when you have all the sprites attached to the group add that group to a play field. Like so.

```
playerGroup->AddSprite(playerSprite);
enemyGroup->AddSprite(enemySprite);

// •
//   additional sprites added
// •

playField->AddGroup(playerSprite);
playField->AddGroup(enemyGroup);
```

The order in which the sprite groups are added to the play field determines the level at which the sprites contained by the group will be drawn. Groups added earlier will appear closer to the background, and later groups will obscure the sprites of earlier added groups.

The code for adding a group to a play field is fairly direct. Add the passed-in group to the play field's group list. Then iterate through the passed list extracting each sprite and adding it to the master sprite list held by the play field.

```
void  CPlayField::AddGroup(CSpriteGroup * newGroup)
{
      ASSERT(newGroup);
      fSpriteGroups->Add(newGroup);

      // Copy the sprite references to the master list
      for(long i = 1; i <= newGroup->GetObjectCount(); i++)
      {
            CSprite *   sprite;

            sprite = (CSprite *)newGroup->GetNthObject(i);
            fAllSprites->Add(sprite);
      }
}
```

Removing Sprites from a Play Field

At some point in a game you might need to replace a whole group of sprites with another. At that point you'll need to use the play field's `RemoveGroup` method. One call and the play field will for-

get all about the group and the sprites it contained. Then you can add another group to the play field and start your game running again.

```
void  CPlayField::RemoveGroup(CSpriteGroup * doa_Group)
{
      ASSERT(doa_Group);
      fSpriteGroups->Remove(doa_Group);

      // Copy the sprite references to the master list
      for(long i = 1; i <= doa_Group->GetObjectCount(); i++)
      {
            CSprite *   sprite;

            sprite = (CSprite *)doa_Group->GetNthObject(i);
            fAllSprites->Remove(sprite);
      }
}
```

The `RemoveGroup` method is the exact inverse of the `AddGroup` method, so no point in spending much time analyzing it. You only need to take notice of one thing with this method. It doesn't do any erasing. That means that if you want the sprites contained in the group you're about to remove not to appear on-screen, you'll need to erase them yourself before calling this method. Otherwise if the sprites were visible and on-screen you would get graphic artifacts whenever a sprite from another group is moved across a sprite belonging to the removed group. "Graphic artifacts" is a nice way of saying your program won't crash, it'll just look like it did.

By using the two group methods together you can build the ability to rearrange the drawing order of the sprites. Remove all the groups from the play field. Sort the groups into your new desired order. Add the groups back into the play field in the new desired order. With the next frame of animation your sprites will appear in their new order.

Performing Sprite Animation

Now that you can create, destroy, and add sprites to a play field at will, it's time to get to the meat. The reason for being here. The real magilla. Animating those sprites. As discussed before, the animation of the sprites takes four stages (ignoring collisions): erasing where the sprites were, drawing the sprites at their new location, and finishing up by copying the sprites from the working buffer to on-screen. I know that makes only three, but let's skip the sprite movement phase for now. All three of these stages are orchestrated by the play field's `ShowNextFrame` method.

```
void  CPlayField::ShowNextFrame()
{
      PixMapHandle      workingPixels;
      PixMapHandle      bkgndPixels;

      // Lock down the pixels
      bkgndPixels = GetGWorldPixMap(fBkgndBuffer);
      workingPixels = GetGWorldPixMap(fWorkplaceBuffer);

      if( LockPixels(bkgndPixels) &&
                              LockPixels(workingPixels))
      {
            // Erase where the sprites have been
            EraseSprites();

            // Draw the sprites where they are now
            BlitSpritesToWorkspace();

            // Finally copy the composited areas to onscreen
            BlitSpritesOnscreen();
      }

      // Free the pixels
      UnlockPixels (bkgndPixels);
      UnlockPixels (workingPixels);
}
```

As you can tell, this method is only a wrapper that dispatches to the play field's internal methods, which do the actual work. Before jumping to the core methods, you need to lock down the

pixels belonging to the workplace and background buffers. These pixels will be unlocked after the animation methods have been called.

This wrapper method can be overridden if you wish to do more than simply animate sprites as part of the animation loop. A good example would be if you would like to have twinkling stars animating in the background. You would create a subclass of the base play field (CTwinklingPlayfield?) and override this method to insert your twinkling code.

Erasing the Sprites

Erasing the sprites is as easy as walking through each sprite group currently attached to the play field, then walking through each sprite in the current group. After the sprite has been gotten out of the current group, it is asked if it needs to be erased. Not every sprite needs to be erased for every frame of animation. If the sprite didn't move or change its visibility from the previous frame, it won't need be erased—or drawn for that matter. Being a good design doobie I've let the sprite decide if it needs to be erased. If the sprite requests erasure it will be asked to give the bounds that the play field should erase. The bounds need to be expressed in the local coordinates of the working and background offscreen buffers. After getting the extent of the impending erasure `EraseSprites` cops out and defers the actual erasing to another method of the play field, `EraseChunk`.

```
void  CPlayField:EraseSprites()
{
      // Point at the working buffer
      SetGWorld(fWorkplaceBuffer, nil);

      // Loop through all the play fields
      for(long i = 1;
            i <= fSpriteGroups->GetObjectCount(); i++)
      {
```

```
                CSpriteGroup * currGroup;

                currGroup =
                (CSpriteGroup *)fSpriteGroups->GetNthObject(i);

                // Loop through all the sprites in the group
                for( long j = 1;
                      j < currGroup->GetObjectCount(); j++)
                {
                      CSprite *   currSprite;

                      currSprite =
                      (CSprite *) currGroup->GetNthObject(j);

                      // Only erase the sprite if it wants to be
                      if(currSprite->ShouldErase())
                      {
                            Rect eraseRect;

                            // Out of each sprite in the group
                            // get the sprite's rect that it
                            // wants erased. The sprite takes
                            // care of calculating this
                            // rectangle as it is moved
                      currSprite->GetEraseRect(&eraseRect);

                            // Got the rect, erase it by
                            // blitting from the background
                            // to the working buffer. Override
                            // this method to provide for
                            // different erasing styles
                            EraseChunk(&eraseRect);
                      }
                }
          }
}
```

The `EraseChunk` method takes the bounds of what you want blitted from the background to the working buffer. The default implementation uses `CopyBits` as its blitter.

```
void  CPlayField::EraseChunk(Rect * blitRect)
{
      CopyBits(   ((GrafPtr)fBkgndBuffer)->portBits,
                  ((GrafPtr)fWorkplaceBuffer)->portBits,
                  &blitRect,
                  &blitRect,
                  srcCopy,
                  nil);
}
```

If you check out the declaration for this method you'll see that is declared as virtual. This is no accident. By replacing this low-level method through the magic of inheritance you can easily drop in your own erasing blitter. And if your hot new blitter finds itself in a bind it will always have a backup in the super classes method.

Drawing the Sprites

Drawing the sprite into the workplace buffer follows the same general outline as the `EraseSprites` method. Loop through the groups. Loop through each sprite in these groups. The only real difference is where the blitting responsibility lies.

```
void  CPlayField::BlitSpritesToWorkspace()
{
      // Point at the working buffer
      SetGWorld(fWorkplaceBuffer, nil);

      // Loop through all the play fields
      for(long i = 1;
                  i <= fSpriteGroups->GetObjectCount(); i++)
      {
            CSpriteGroup * currGroup;

            currGroup =
            (CSpriteGroup *)fSpriteGroups->GetNthObject(i);

            // Loop through all the sprites in the group
            for( long j = 1;
                  j < currGroup->GetObjectCount(); j++)
```

```
            {
                CSprite *   curSprite;

                curSprite =
                (CSprite *) currGroup->GetNthObject(j);

                // Only blit the sprite if it wants to be
                if(curSprite->ShouldDraw())
                {
                    curSprite->BlitFrame(fWorkplaceBuffer);
                }
            }
        }
}
```

Instead of calling a low-level method of the play field to blit the sprite onto the working buffer, this method relies on the sprite itself to handle the blitting of its frames. The sprite's `BlitFrame` method only needs to be passed in the GWorld that you want the sprite to be blitted to. This method is where you get to apply all that masked-blitting knowledge that you so dutifully crammed into your frontal lobes.

Like the play field's `EraseChunk` method, the `BlitFrame` method was designed to be replaced. When you need more speed you simply override the base call and drop in your own masked blitter. The details of this method and overriding it will be discussed in the upcoming sprite chapter.

From Offscreen to On-screen

At this stage the workplace buffer has exactly the image you would like on-screen. Good thing the play field has a method for copying the working buffer to the screen. After focusing QuickDraw at the play field's host window, the `BlitSpritesOnscreen` method follows the same looping through sprites and groups pattern as the previous animation stages. In order to save time this method only copies over the sprites that need to be copied. This is determined by asking the sprite through its `NeedCopiedOnscreen` method.

```
voidCPlayField::BlitSpritesOnscreen()
{
    // Point at the host window
    SetPort(fHostPort);

    // Loop through all the play fields
    for(long i = 1;
                i <= fSpriteGroups->GetObjectCount(); i++)
    {
          CSpriteGroup * currGroup;

          currGroup =
          (CSpriteGroup *)fSpriteGroups->GetNthObject(i);

          // Loop through all the sprites in the group
          for( long j = 1;
                j < currGroup->GetObjectCount(); j++)
          {
                CSprite *   currSprite;

                currSprite =
                (CSprite *) currGroup->GetNthObject(j);

                // Only copy onscreen if need to
                if(currSprite->NeedCopiedOnscreen())
                {
                      Rect copyRect;

                      // Need to move erased and freshly
                      // drawn sprite onscreen. The erase
                      // rectangle will return both
                      currSprite->GetEraseRect(&copyRect);

                      // Use low-level onscreen blitter
                      CopyChunkOnscreen(&eraseRect);

                      currSprite->MarkAsOnscreen();
                }
          }
    }
}
```

The bounds of the upcoming pixel copy are retrieved by asking the sprite for its erasing bounds, which matches the pixels that need be copied from the working buffer to the host window. The actual

pixel blasting is again performed by a lower-level method of the play field. After the blitting, the sprite is informed that it has been successfully displayed on-screen. After calling this method the sprite will tell future passes through the animation loop to ignore this sprite. Unless, of course, something happens to the sprite that would require its on-screen persona to be updated, moving being a good example.

```
void CPlayField::CopyChunkOnscreen(Rect * copyBnds)
{
      CopyBits(   ((GrafPtr)fWorkplaceBuffer)->portBits,
                  ((GrafPtr)fHostPort)->portBits,
                  &copyBnds,
                  &copyBnds,
                  srcCopy,
                  nil);
}
```

The core method `CopyChunkOnscreen` uses `CopyBits` as its default blitter. Like the play field's `EraseChunk` method, `CopyChunkOnscreen` is meant to be replaced if you desire a faster blitter. As this method moves the composited sprites on-screen, any replacement method will have to follow all the proper precautions of drawing directly to the screen. These all-important rules of on-screen drawing will be discussed later.

Moving the Sprites

With all those blitting steps out of the way, you can now look at the fourth, and final, stage of animating your sprites: the play field's `MoveSprites` method. First off, the method's name doesn't describe it fully. While `MoveSprites` does move the sprite around the play field, it has the added responsibility of coordinating the changing of the sprite's internal animation frames. I felt that the two tasks were linked together and should be performed at the same time and that the movement of the sprite was the dominant task of the method; hence its name. Who cares what the method is called? Let's check out the code.

```
void CPlayField::MoveSprites()
{
      // Loop through all the sprite groups
      for(long i = 1; i <=
                  fSpriteGroups->GetObjectCount(); i++)
      {
            CSpriteGroup * currGroup;

            currGroup =
            (CSpriteGroup *)fSpriteGroups->GetNthObject(i);

            // Loop through all the sprites in the group
            for( long j = 1;
                  j < currGroup->GetObjectCount(); j++)
            {
                  CSprite *   currSprite;

                  currSprite =
                  (CSprite *) currGroup->GetNthObject(j);

                  if(currSprite->IsTimeToMove())
                        currSprite->Move();

                  if(currSprite->IsTimeToChangeCels())
                        currSprite->ChangeCel();

            }
      }
}
```

Like all the other methods that work with the sprites, this method spends its time looping through groups to get down to the individual sprites. Before rushing right ahead and moving the sprite, the play field must verify that it is time to move the sprite through the sprite's `IsTimeToMove` method. Not every sprite will need to be moved on every cycle of the game loop; therefore the need to check before moving the sprite. If the sprite says its O.K. to move it, then the play field turns right around and asks the sprite to get up off its fat butt and move itself.

The same sequence of asking the sprite and then telling the sprite to do it itself is performed for cycling through the sprite's various image cels. Like the sprite movements the cels will move at

a rate that isn't guaranteed to match the game loops, and the play field needs to ask permission before forcing the sprite to cycle its current animation cel.

If you said to yourself. "Hey, Self, this guy must be an idiot. How can the program be in a tight loop, like this game loop thing, and ever have the method `IsTimeToMove` change state? There's only one processor and it's busy doing all this animation. I sure can't see where the sprite's getting the time to figure out it needs to be moved let alone the time to check whether its cel needs changing." Good question and stop calling me an idiot.

The answer to that question is that the sprite's code that determines whether it's time to move the sprite or cycle to its next cel is run as a time manager task. And since these tasks run at interrupt level they are running asynchronously—a big word that means not running at the same time—of the game loop. In the chapter on sprites and sprite cels you'll be up to your knees in this time manager stuff so for now ignore the implementation details. I do this by pretending that it's done by magic. Keeps my life simpler and conversations shorter.

Checking for Collisions

During your game you'll want to use `CheckForCollisions` to ask the play field to check for any possible sprite collisions with the play fields. You usually want to administer the collision test right after moving the sprites and before showing the next frame of animation.

The play field's only role in collision checking is as a starting point. All it does is loop through each of the sprite groups and ask them to check for any collisions they might have had. Each sprite group has a list of other sprite groups that they are, according to the rules of the game, allowed to collide with. You tell each group what other groups to collide against when you are creating the groups for the play field.

```
void CPlayField::CheckForCollisions()
{
            // Loop through all the play fields
      for(long i = 1;
                  i <= fSpriteGroups->GetObjectCount(); i++)
      {
            CSpriteGroup * currGroup;

            currGroup =
            (CSpriteGroup *)fSpriteGroups->GetNthObject(i);
            currGroup->CheckForCollisions();
      }
}
```

Like the other core methods of the play field, `CheckForCollisions` can be overridden. If you need to use one of the more sophisticated collision-detection methods go right ahead. None of the other code for the play field knows or cares how the collision testing is performed. Override at will.

Play Field Miscellany

The only thing left on the play field discussion table are condiments. The tiny little methods that you'll need but are only supplied for convenience. Luckily, there only a few of these methods so you can quickly read this section and move on to the next chapter.

Handling Update Events

When you stop your game and are outside of the game loop and back in the normal event loop, you'll need to handle any update events that the Mac passes out to you. You could handle the update event by calling the method `ShowNextFrame`, except that method advances the game one sprite cel. Another problem would be that `ShowNextFrame` expects to be able to update correctly for every

sprite that moved. This might not be possible because the update region clips the host window. If the update region intersects a sprite's movement bounds, then only the part of the sprite that lies within the region will be updated. Your update will have trashed your screen.

Now, in your quest for speed you might have upgraded the default working buffer to on-screen blitter to completely ignore the windows' various clipping regions (visible, update, and clip). This would eliminate the sprite breakup due to the update region, but if your window is overlapped by any other window you'll end up spraying your pixels all over someone else's window. Not a neighborly thing to do.

To prevent these and other update event *faux pas*, the play field has a method strictly for dealing with update events, creatively named `HandlePlayFieldUpdate`. Pass it the rectangle that surrounds the area in need of updating and you're done.

```
void CPlayField::HandlePlayFieldUpdate(Rect * updateBounds)
{
      SetPort(fHostPort);

      CopyBits(   ((GrafPtr)fWorkplaceBuffer)->portBits,
                  ((GrafPtr)fHostPort)->portBits,
                  &copyBnds,
                  &copyBnds,
                  srcCopy,
                  nil);
}
```

This method won't win any award for ingenuity. But it works and that's all you need. Don't try speeding this method up. It isn't worth the effort. You should only be receiving update events in areas of your game that are not time-critical. Like when the game is paused, or when it's in the background. Basically anytime you're not in a tight game loop. Your update handling code should smell something like this example.

```
                                        .
                                        .
                                        .
case updateEvt:
{
      WindowPtr   updateWind = (WindowPtr)event->message;
      WindowPeek  windInards = (WindowPeek)event->message;
      Rect         bnds;

      bnds = *((windInards->updateRgn))->rgnBBox
      SetPort(updateWind);
      BeginUpdate(updateWind);
            myPlayField->HandlePlayFieldUpdate(&bnds);
      EndUpdate(updateWind);
}
break;
                                        .
                                        .
                                        .
```

One word of caution. Actually a sentence of a caution. What would one word of caution be, anyway? "Duck!" The sentence of caution: Don't perform an update in the middle of a game loop. You want the whole move, collide, and animate cycle to complete. Otherwise you could end up with a workplace buffer that is out of synchronization with the game, and when you drop back to the game loop your screen will be trashed.

Drawing in the Offscreen Buffers

At the start of your game, or at the start of a level, you'll want to establish the background image your sprites will be moving across. The easiest way would be to have QuickDraw render your background with a call to `DrawPicture`. Your problem, and the reason for these paragraphs, is that you get QuickDraw focused at the background buffer. As both offscreen buffers of the play field are declared protected and therefore invisible to any code outside of the play field and its descendants, you'll need some assistance from

the play field to get to those shy buffers. That assistance is provided in these four methods.

```
void  PreDrawOnBackground();
void  PostDrawOnBackground();

void  PreDrawOnWorkplace();
void  PostDrawOnWorkplace();
```

Using these methods follows the same pattern as `BeginUpdate` and `EndUpdate`. Prepare the buffer for QuickDraw drawing. Go crazy with QuickDraw. Turn off drawing. The indenting of `PaintRect` is just a style I use. I like to indent any code that is bracketed by two phases like these two methods.

```
Rect  bkgndBnds;

myPlayField->GetBounds(&bkgndBnds);

// Paint background with a nice gray pattern
myPlayField->PreDrawOnBackground();
      PaintRect(&bkgndBnds, &qd.grey);
myPlayField->PostDrawOnBackground();
```

On the rare occasion that you need to draw into the working buffer, use the pattern but replace the background focusing methods with their twin working buffer ones.

```
void  CPlayField::PreDrawOnBackground()
{
      ASSERT(fBkgndBuffer);

      GetGWorld(&fOldCPort, &fOldGDevice);
      SetGWorld(fBkgndBuffer, nil);
}
```

The buffers are prepared for drawing by first saving the current port and graphic device that QuickDraw is now using. The port and device are saved into reference variables owned by the play field. These variables are used by both sets of methods. While this is convenient for you it also means that you don't want to nest calls to

these methods. No calling `PreDrawOnWorkplace` after or between calls to `PreDrawOnBackground` and `PostDrawOnBackground`.

```
void  CPlayField::PostDrawOnBackground()
{
      SetGWorld(fOldCPort, fOldGDevice);
}
```

The method `PostDrawOnBackground` cleans up after your offscreen drawing adventures by simply restoring QuickDraw's previous focus.

Drawing in the Host Window

If you want to draw in the host window and have somehow managed to misplace your window pointer (I'm always doing that), you'll be relieved that you can quit looking for it and just ask your play field what window it is attached to.

```
CGrafPtr    GetHostGrafPort() { return fHostPort;}
```

This method is declared inline for those rare times when you have to call it thousands of times in a row.

Play Field Summary

Nothing quite says chapter summary like a full dump of the play field's class definition.

```
class CPlayField {
public:
                CPlayField(CWindowPtr * hostWindow,
                                const Rect* hostBounds);
      virtual ~CPlayField();
      virtual void     MoveSprites();
```

```
        virtual     void CheckForCollisions();
        virtual     void ShowNextFrame();

                    void AddGroup(CSpriteGroup * newGroup);
                    void RemoveGroup(CSpriteGroup * doa_Group);

                    void PreDrawOnBackground();
                    void PostDrawOnBackground();

                    void PreDrawOnWorkplace();
                    void PostDrawOnWorkplace();

        virtual     void
              HandlePlayFieldUpdate(Rect * updateBounds);

              CGrafPtr   GetHostGrafPort()
                              { return fHostPort;}
                    void GetBounds(Rect * playFieldBnds)
                              { return fPlayfieldBnds; }

protected:
        GWorldPtr           fBkgndBuffer;
        GWorldPtr           fWorkplaceBuffer;
        CGrafPtr            fHostPort;
        Rect                fHostBnds;
        Rect                fPlayfieldBnds;
        CObjectList *       fSpriteGroups;
        CObjectList *       fAllSprites;

              void  CreateGWorld(GWorldPtr * resultGWorld,
                                          Rect * bounds);
                    void EraseSprites();
                    void BlitSpritesToWorkspace();
                    void BlitSpritesOnscreen();

        // Core routines that blit.
        // Default version use CopyBits

        virtual     void EraseChunk(Rect * blitRect);
        virtual     void CopyChunkOnscreen(Rect * copyBnds);

private:
        GDHandle            fOldGDevice;
        CGrafPtr            fOldCPort;
};
```

9

Sprites and Sprite Cels

To get sprites leaping and frolicking across the screen, you'll have to understand the relationship between sprites and sprite cels. It's a simple relationship. Sprites take the play field's request for sprightly actions and either the sprite cel either really performs the task or the sprite schedules the task to be performed at a more appropriate time. Sprite cels are the containers for the sprites' animation frames and masks. With the sprite cel hiding the details of how the sprite is stored and how the cel is blitted, the sprite is freed from the tedious responsibility of doing any work.

Creating a Sprite

You can either create a sprite raw, forcing you to embed all the sprite's attributes within your code, or you can create the sprite from a resource template, allowing you to change the sprite's settings without resorting to a recompile of your game. Guess which method I prefer.

```
CSprite::CSprite(SpriteID spriteID)
{
      SpriteX();
      fDogTag = spriteID;
}
```

Before going into the template method let's get the ordinary constructor out of the way. This sprite constructor takes a four-character OSType as its only argument. This tag will be used to properly identify the sprite. Tagging your sprites will help in the future when you need to see what type of sprite has collided with your sprite. Other than assigning the tag to the object's dog tag field, this constructor's only other duty is to call the sprite's private initialization function, `CSpriteX`.

This method initializes the sprite to a known state for the sprite's destructor. If you add any other type of constructors to this object make sure they make a mandatory call on `CSpriteX` before moving on and doing anything that might bring the sprite's destructor into action.

This constructor is of little practical use with this class. Its real purpose is to provide a construction stage for any descendants of the sprite class that don't use the sprite cels to accomplish their on-screen rendering. A good example of such a situation would be a sprite that showed the player's current score. This sprite would only need to override the sprite's drawing method and replace it with code that drew the current score as a text string. Since this type of sprite would not be using the sprite cels, it would want to use this constructor in order to prepare the sprite. A later game example will use this exact tactic to display the player's score.

Sprite Templates

To create a sprite from template install the sprite's resource template (the resource type of the template is 'SpTm') into your favorite resource editor, or copy the template into your game's resource file. If you're more the command line-oriented type you can use the Rez file provided. It's sitting right next to the ResEdit template on the source disk.

To create a sprite you'll first need to fill out a sprite ("SpTm") template. For example, the sprite has an id of "Rock," a starting position of 100, 150, a movement delay of 20 milliseconds, a cel change delay of 25 milliseconds, its visibility set to true, a starting index of 1, and an array of 2 color icon ids starting at the number 3000. With this resource template the sprite's constructor can build a fully functioning sprite.

```
CSprite::CSprite(short templateID )
{
      OSErr            err;
      SpriteTempHand templateH;
      short            index;

      SpriteX();

      // Get the sprite load template resource
      templateH = GetResource(kSpriteTemplate, templateID);
      if(templateH == nil)
      {
            PostFatalError(ResError());
            return;
      }

      // Create each cel and add it to the cel list
      err = noErr;
      HLock((Handle) templateH);
      for(short i = 0; i < (*templateH)->celCount; i++)
      {
            CSpriteCel * cel;

            cel = new(CSpriteCel, (*templateH)->cicnIDs[i]);
            fCels->Add(cel);
      }
```

```
        // Fill in the rest of the fields
        fDogTag = (*templateH)->id;
        fVisible = (*templateH)->visible != 0;
        index = (*templateH)->celIndex;
        if(index > (*templateH)->celCount)
              index = 1;

        moveTime = (*templateH)->moveDelay;
        celChangeTime = (*templateH)->celDelay;

        HUnlock((Handle) templateH);
        ReleaseResource((Handle) templateH);

        // Mark the cel list as used
        fCels->IncrementRefCount();

        // Show correct cel
        SetCurrentCel(index);

        // Fire up them timers
        SetAutoMoveTime(moveTime);
        SetCelCycleTime(celChangeTime);
}
```

The constructor takes the resource id of the sprite template for this sprite as its only parameter. After scrubbing the sprite clean in case of an unexpected creation failure, the constructor loads in the sprite template. For each color icon listed in the template a matching sprite cel is created. The cels are added to the cel list maintained by the sprite. Once all the sprite's cels have been built, the rest of the sprite's fields are initialized to the template's matching values. After the template is stripped of its useful information it is quickly unlocked and discarded.

The sprite's cel list needs to have its reference count incremented to reflect that at least one sprite is currently using the cels contained in the list. This will make more sense later in the section that talks about reusing sprite cels.

The sprite is requested to use the starting cel specified by the template with a call to `SetCurrentCel`.

Finishing up the sprite's construction are the calls to `SetAutoMoveTime` and `SetCelCycleTime`. These two calls fire up the timers that tell the sprite when enough time has elapsed and it can

safely move the sprite, or, in the case of `SetCelCycleTime`, when it is time to change to the next cel owned by the sprite. Both functions are passed the number of milliseconds that the timers should delay between actions.

Building Sprite Cels

Before moving on to the other responsibilities of sprites, let's take a quick side journey into the sources of sprite cels. A sprite cel is a fairly simple class with only a few public responsibilities to worry about: creation, destruction, blitting the cel's image to a graphic world, and determining if it and another sprite cel are intersecting.

```
class CSpriteCel {
public:
                  CSpriteCel();
                  CSpriteCel(short cicnID);

      virtual     ~CSpriteCel();

      virtual     void  BlitToBuffer(GWorldPtr buffer,
                                     const Rect * target);

      virtual Boolean   Intersect(CSpriteCel * testCel,
                        const Rect * testCelBnds,
                        const Rect * myBnds);

protected:
      GWorldPtr         fCelImage;
      RgnHandle         fMaskRgn;
      PixMapHandle      fCelPixels;
      Rect              fCelBnds;
      short             fHorzMaskOffset;
      short             fVertMaskOffset;

      OSErr MakeRgnMask(CIconHandle iconH);
      OSErr MakeCelGWorld(CIconHandle iconH);

private:
      void  CSpriteXCel();

};
```

Sprite cels are the containers for the sprite's imagery and masks. Each sprite cel holds one frame of the sprite's imagery and that image's matching mask. The sprite, and the rest of the program for that matter, has no idea how the pixels making up the sprite and mask are stored. All it knows is that it has a list of sprite cels. With the sprite cel being the only class aware of how the sprite's image and mask are stored, it is also given the job of blitting that image whenever requested.

While the cel does have a default constructor, the one we're most interested in is the constructor that takes a color icon's resource id. This is the constructor that the sprites you create will be using the most. The default constructor is provided for the more esoteric descendants of sprite cels.

```
CSpriteCel::CSpriteCel(short cicnID)
{
      CIconHandle iconH;

      CSpriteXCel();

      // Load in the Color icon
      iconH = GetCIcon(cicnID);
      if(iconH)
      {
            OSErr err;

            HLock((Handle) iconH);

            // Create the cel's pixel image
            err = MakeCelGWorld(iconH);

            // Turn icon's mask into a region
            if(!err)
                  err = MakeRgnMask(iconH);

      // Before posting potential error get rid of the icon
            DisposeCIcon(iconH);

            if(err)
                  PostFatalError();
      }

}
```

The sprite cel is prepared for potential destruction with the starting call to the sprite's private helper function, `CSpriteCelX`. Like the sprite's helper function, the cel's prepares the cel for a potential call of the cel's destructor. And again like the sprite's constructors, any added constructors you might tack onto this class should make a call to this helper function before doing anything else.

After scrubbing the cel clean for the destructor, the color icon specified by the passed argument is loaded into memory with a call to the Toolbox routine `GetCIcon`. If the color icon exists and there is enough memory to load it in (keep your fingers crossed), you'll be handed back a handle to a color icon structure.

```
struct CIcon {
      PixMap      iconPMap; /*the icon's pixMap*/
      BitMap      iconMask; /*the icon's mask*/
      BitMap      iconBMap; /*the icon's bitMap*/
      Handle      iconData; /*the icon's data*/
      short       iconMaskData[1];
      /*icon's mask and BitMap data*/
};
```

This structure will contain the pixel map that holds the cel's future image and a bitmap representing the icon's mask. It also has a bitmap holding a black-and-white version of the color icon that this type of sprite cel ignores. If you wanted to support black-and-white screens in your game you could easily add a descendant of the sprite cel that uses the black-and-white bitmap of the icon in addition to the color version.

Next, a graphic world is created that holds a copy of the color pixels contained in the icon with a call to the cel's protected method `MakeCelGWorld`. This method creates a graphic world and copies the pixels from the color icon to the pixel map contained within the graphic world.

```
OSErr CSpriteCel::MakeCelGWorld(CIconHandle iconH)
{
```

```
        OSErr err;
        Rect  celBnds;

        ASSERT(iconH);

        celBnds = (*iconH)->iconPMap.bounds;

        // Fix the gworld's coordinates
        OffsetRect(&celBnds, -celBnds.left, -celBnds.top);
        err = MakeGWorld(&fCelImage, &celBnds);
        if(!err)
        {
              CGrafPtr          oldPort;
              GDHandle          oldDevice;
              PixMapHandle      celPixels;

              GetGWorld(&oldPort, &oldDevice);
              SetGWorld(fCelImage, nil);
              celPixels = GetGWorldPixMap(fCelImage);
              if(celPixels != nil && LockPixels(celPixels))
              {
                    // Blit the pixels
                    PlotCIcon(&celBnds, iconH);

                    // Save the pixel handle.
                    // Cel's pixel handle will stayed locked
                    // until its destruction
                    fCelPixels = celPixels;
              }

              SetGWorld(oldPort, oldDevice);
        }

        return err;
}
```

The sprite cel needs to have dimensions that match the icons but with the buffer's top-left coordinates at 0,0. Thus the reasoning for the `OffsetRect` of the bounds rectangle copied from the icon pixel map bounds. Using the freshly normalized bounds rectangle, `MakeGWorld` will attempt to create an offscreen buffer for the sprite cel. If it is successful, the pixels of the world will be locked and the icon's color image transferred through the Mac's

PlotCIcon procedure. Everything after the icon plotting is simply closure.

You'll notice that as part of copying the icon's pixels the graphic world's pixels are locked with a call to LockPixels and that they are never unlocked. This isn't a bug. It's an optimization. The pixels are kept locked for the entire lifespan of the sprite cel. The one-time locking is done so that the cel's drawing code can avoid the costs of locking and unlocking the pixels before and after every blit. This simplifies and speeds up the cel's implementation at the cost of potential heap fragmentation. A fair trade-off, as most of your cels will probably be brought in at the beginning of the game where the potential for fragmentation will be minimized.

To complete the sprite color icon constructor, the cel's mask region needs to be extracted from the icon's mask bitmap. The sprite cel's utility method MakeRgnMask will perform that exact trick.

```
void  CSpriteCel::MakeRgnMask(CIconHandle iconH)
{
      BitMap      maskBits;
      OSErr err;

      ASSERT(iconH);
      ASSERT(fMaskRgn);

      HLock((Handle) iconH);
            maskBits = (*iconH)->iconMask;
            // Patch up the bitmap's address and bounds
            OffsetRect(&maskBits.bounds,
                              -maskBits.bounds.left,
                              -maskBits.bounds.top);
            maskBits.baseAddr =
                              (Ptr) (*iconH)->iconMaskData;
            // Make a mask
            err = BitMapToRegion(fMaskRgn, &maskBits);
      HUnlock((Handle) iconH);

      // Remember the delta from the mask's pixels upper
      // left to the region's upper left. We'll need this
      // info to properly reposition the mask relative to
      // cel's blitting destination
      if(!err)
```

```
        {
                fHorzMaskOffset = (*fMaskRgn)->rgnBBox.left;
                fVertMaskOffset = (*fMaskRgn)->rgnBBox.top;
        }

        return err;
}
```

A bitmap structure pointing at the 1-bit-deep image date contained within the color icon is generated. Again the bounds rectangle is normalized so that the upper left is at 0,0. After you build the bitmap the truly cool Toolbox routine `BitMapToRegion` is used to transform the mask bitmap into a region. This nifty function will either return a successfully built region or an error, with the possible choices being `pixmapTooDeepErr` (–148) or `rgnTooBigErr` (–500). The first result means you handed `BitMapToRegion` a pixel map that is deeper than 1 bit, not easy to do in this method. The other result means that the generated region would be larger than QuickDraw allows (64K). If you get this error you must have created a monster of a sprite, probably one too hairy to be of any practical use. Fire up that copy of SuperPaint—the official painting program of *Sex, Lies, and Video Games*—and make a simpler sprite. Any other problems than these two you don't need to worry about, as QuickDraw will have crashed and you'll be wondering who this Jackson guy is and why his name shows up whenever QuickDraw decides to take a dirt nap.

Before returning the mask region the code needs to stash away the offsets from the mask bitmap's upper-left corner to the upper-left corner of the resultant region. You need to do this because the region could end up being smaller than the bitmap and when you move the region to match the on-screen position of the sprite, you'll need to adjust for this difference in sizes. Otherwise the region will be incorrectly aligned around the sprite cel's upper-left corner.

With sprite cel creation having been fully dissected under the harsh halogen lights of this section, it's now time to look at the only other way of creating sprites. Cloning. Boy, that sure sounds ominous.

Cloning a Sprite

Sprite cloning is not done by binding down the original sprite on one half of a giant lazy Susan and placing a lump of gray clay-looking stuff on the other side and spinning the plate at torturous speeds until the lump gains all the characteristics, including the bad toupee, of the original sprite. This never works. The source sprite always keeps repeating something about "stupid half-breed Vulcan" screwing up the process. Too bad though, that would be much cooler than how you'll do it.

```
CSprite * clonedSprite;

clonedSprite = sourceSprite->Clone();
if(clonedSprite == nil)
      PostFatalError(memFullErr);
```

One call and some error checking and you'll have successfully cloned a sprite. Now comes the tricky part—the clone created would not match a sprite created from scratch. Cloned sprites are memory-optimized versions of the original sprites.

The reasoning behind sprite cloning is that most games have several sprites that are duplicates of each other except for their on-screen locations. Think of a game like Asteroids. Each one of those chunks of space debris looks exactly like the other floating chunks with the only distinguishing feature being that one chunk is in the upper-left corner and the other closer to the center of the screen ripping your ship apart. With several identical sprites on-screen at once you would be chewing up gobs of memory (*gobs* comes right after kilobytes and before megabytes) for each sprite. And each sprite would be holding sprite cels that are exact duplicates of the other sprite's cels. Wouldn't it be nice if all the sprites could share the same sprite cels between themselves instead of wasting all that precious memory? Yes, it would, and that's why sprites have a cloning method.

```
virtual   void *Clone() { return new(CSprite,(*this)); }
```

The `Clone` function of a sprite class is a simple one-line inliner that calls `new` passing the source sprite you would like cloned. The `new` operator will end up calling the sprite's copy constructor, passing it along the original sprite.

```
CSprite::CSprite(CSprite & source )
{
      SpriteX();

      fDogTag = source.fDogTag ;
      fCurrBnds = source.fCurrBnds;
      fPrevBnds = source.fPrevBnds;
      fEraseBnds = source.fEraseBnds;
      fCurrWidth = source.fCurrWidth;
      fCurrHeight = source.fCurrHeight;
      fMoveExtent = source.fMoveExtent;

      fEraseMe = source.fEraseMe;
      fDrawMe = source.fDrawMe;
      fVisible = source.fDrawMe;

      fTimerMoveFlag = source.fTimerMoveFlag;
      fTimerCelFlag = source.fTimerCelFlag;
      fMoveHoriz = source.fMoveHoriz;
      fMoveVert = source.fMoveVert;

      fCels = source.fCels;
      if(fCels)
            fCels->IncrementRefCount();
      fCurrentCel = source.fCurrentCel;
      fCelIndex = source.fCelIndex;

      fMoveDelay = source.fMoveDelay;
      fCelDelay = source.fCelDelay;
}
```

Like any good sprite constructor, the copy constructor calls the sprite's private initialization function, `CSpriteX`. From there the constructor runs through all the fields of the source sprite, copying them over to the newly built clone. Copying over the cel list reference (`fCels`) is where the memory saving takes place.

If the source sprite bothered to have a live cel list, that reference is copied over to the cloned sprite's cel list reference. The cel

list now has two sprites currently referencing it. To reflect that fact the copy constructor asks the cel list to up its internal reference count by one with a call to the cel list's `IncrementRefCount` function.

Now that the original sprite cels have been cloned in a memory-efficient manner, the copy constructor finishes up by copying over the rest of the source sprite's internal fields.

For the cost of a new sprite object the `clone` function has returned a duplicate of the source sprite that can then be used as an independent sprite without your having to worry about wasting any more priceless memory.

Cel Lists

Sprite cel lists are descendants of our standard list class that have a few added functions that keep track of how many sprites have referenced the list.

```
class  CCelList   : public CObjectList     {
public:
      CCelList() { fRefCount = 1};
      ~CCelList();

      void  IncrementRefCount() { fRefCount++; }
      void  DecrementRefCount() { fRefCount--; }
      short GetRefCount() { return fRefCount; }

protected:
      short fRefCount;
};
```

Upon construction the cel list has its reference count set to one, based on the thinking that if someone bothered to create a cel list they must want to use it and they'll want the reference count to reflect that.

Destruction of the list just performs a safety check before letting the real destruction take place. The safety check verifies that the reference count is one or below. A reference count above one at de-

struction time would be bad. A reference count greater than one would imply that a sprite is still out there that thinks this cel list is valid. On that uninformed sprite's next access to its sprite cel list it will find a nasty surprise. No list, just memory that looks like a list once lived there. You don't want that, hence the safety check.

Cel List Reuse

Reuse of the cel list comes through the proper use of the cel list's other three functions. These functions allow the sprite to maintain the usage counts of the list.

Each time a sprite copies over its cel list it needs to call `IncrementRefCount`. Which does exactly what its name says. Whenever the sprite is done using a sprite list it will use the other two functions: `DecrementRefCount` and `GetRefCount`. Again their names describe their entire implementations.

A design problem with the cel lists is the dependency on the sprite to do the right thing with the lists. Depending on one object to correctly use another always leaves room for disaster. How much room does a disaster take up, anyways? A better design would have the whole reference counting scheme hidden from the sprite. This would require trickier C++ code than I wanted to use for this book, things like operator overloading and such.

Anyway, let's get back to the real subject of this chapter, sprites.

Disposing of a Sprite

Having created a sprite, at some later time you might want to dispose of the darn thing. Easy enough, just use the `delete` operator or if the sprite is attached to a play field through a sprite group, the sprite will be disposed of when the play field is destroyed. Either way, the sprite's destructor will eventually will be called.

The sprite's destructor has only two tasks to worry about: take care of the sprite cel's under the sprite's control, and shut down the timers used by the sprite.

```
CSprite::~CSprite()
{
      if(fMoveTimer.tmAddr)
            RmvTime((QElemPtr)&fMoveTimer);
      fMoveTimer.tmAddr = nil;

      if(fCelTimer.tmAddr)
            RmvTime((QElemPtr)&fCelTimer);
      fCelTimer.tmAddr = nil;

      if(fCels && fCels->GetRefCount() <= 1)
      {
            fCels->FreeAll();
            (fCels);
      }
      else
            fCels->DecrementRefCount();

      fCels = nil;
}
```

Making sure that the sprite's timers were installed with a quick check of the timer's function pointer, we remove both sprite timers from the Mac's time manager's consideration with a call to `RmvTime`. In the spirit of paranoia the timer's function pointers are cleared out.

With the sprite's timers shut down the destructor turns its attention to the sprite's cels. The sprite should only dispose of the sprite cels if the reference count returned by the list's `GetRefCount` function returns one or less. If the list gives the all-clear signal the cels contained in the list are released back into the memory pool and the list itself is then wiped from the sprite's memory. If the list indicates that it is still in use by some other sprites through a reference count greater than one, then the destructor's only responsibility is to knock the list's reference count by one with a swift call to `DecrementRefCount`. One of the other sprites sharing the cel list will have the pleasure of wiping out the cels and list when its destructor finally gets called. Maintaining our paranoia, the sprite cel list reference is zeroed out.

Sprite Timers

Now that you can create and destroy sprites like some sort of game-programming demi-god, it's time to start moving those sprites. The first thing you need to know about sprite movement is timing.

In order for your sprites to move around on the screen they have to have a sense of time. Your sprites must move so many pixels in a certain amount of time. The question is, what provides that time? Sounds like a simple question, doesn't it? What is time? In your case time can be established from a couple of sources. One is the game's animation frame rate. In this case, your sprite would move so many pixels per frame of animation. The other basis of timing could be wall-clock time, or time as it exists in the world independent of the game. Here, your sprite would move so many pixels per second or, more realistically, so many pixels per millisecond.

Timing your sprite's movements through the game's frame rate is an easy way to provide a clocking source for your sprites. Inside your game and before or after each screen update you simply ask all the sprites to move. Each sprite will know how many pixels and in what direction and they will trudge along a little for each frame of animation generated. Simple enough. So simple that you know that there has to be problem with it. The problem arises when the frame rate of the game varies. If the frame rate slows down, all of the sprites slow down with it. Move your game to some parallel processing monster Mac of the future and your sprites will be skittering across the screen like a five-year-old filled to the gills with chocolate-covered expresso beans.

The too fast case is pretty simple to fix. Just throttle the frame rate back. No matter how fast the hardware is capable of blasting up frames of animation, don't let it. Cut it off at some playable level—thirty frames per second is a good choice. The hard case is hardware that is slower than average. Here you're stuck. Your only solution is to try and speed up the frame rate as much as possible on these cycle-challenged processors.

The other method of timing your sprites is with a clock that is external to the game. In this scenario, you would establish a movement distance per unit of time that your sprite would move; say, two pixels every twenty milliseconds. Since the timing for the sprite is independent of the game, your sprites will move two pixels

every 20 milliseconds no matter what the speed of the host processor is. To accomplish this temporal feat, you need some way for your sprite to know when it's time to move. The best way for you to do this would be for each sprite to establish a timer that counts down the appropriate number of milliseconds; when the timer expires, your sprite shuffles itself over slightly. And, of course, you would want the timers to run parallel with the rest of the game.

As you've seen from the previous sections, establishing a timer per sprite is the timing approach that the class kit takes. Actually, two timers are set up: one for timing the sprite's movements and another for controlling the rate of change for the sprite's cels. The guts of these sprite timers are based on the Mac's built-in timing mechanism, cleverly named the Time Manager (see *Inside Macintosh: Processes*). This chunk of ROM code gives us almost exactly what we want.

Using the time manager to set up a timer is fairly straightforward. Create a time manager structure. Fill it out and register it with the Time Manager with a call to its **InsTime** function. To start the timer off you now only need to call the Time Manager's **PrimeTime** function. This function takes a pointer to the time manager structure and the number milliseconds that you want the timer to count down. When that number of milliseconds has elapsed, you will be notified via the function pointer you thoughtfully installed in the time manager structure as part of filling it out. To kill off the timer, make a call to **RmvTime**, passing it the address of the timer structure, and it will gladly eradicate your timer from existence—both spatially and temporally.

The fun part of dealing with the Time Manager is when it calls the function you supplied as part of the time manager structure. Your function will be called when the timer expires. Inside your function you would like to move the sprite along or have it move on to displaying its next cel. That's what you would like to happen. Dream on. When your supplied timer function is called, you are operating at interrupt level. When a Mac is at interrupt level you can't do much, and you have to do it fast. Almost any Toolbox routine you would like to call will, if you're lucky, immediately crash your Mac. At interrupt time you can call only Toolbox routines that *Inside*

Mac has declared as safe. Nothing in QuickDraw is safe. Nothing that is any fun is safe.

The standard technique for dealing with this dilemma is to have your interrupt service routine simply set a flag that the timer has expired and boogie on out of there. Later, while at a non-interrupt level, your main thread of the application can check this flag and then perform, with impunity, the functions it would have preferred to have done at interrupt time.

In our case the timers installed by the sprite set flags internal to the sprite when they expire. Later, when it's safe, during the game loop, the flags are checked and if they have been set by the timers the sprite is asked to move along or cycle to the next cel in its series. After the sprite is moved, its cel is changed, the timers are reset, and the whole process starts over again.

With the nature of time out of the way, you are now free to head on to sprite movement. Have fun.

Moving a Sprite

There are two ways to move your sprites around on the screen, automatically and by hand. Each has its advantages and disadvantages. Automatic movement is a great way to "fire and forget" a sprite's movements but doesn't allow for very complex movements. The opposite method of sprite movement requires you to do all the work in moving the sprite. With your code in control of all sprite movements, it has absolute flexibility in what your sprites do on-screen.

Setting a sprite to move automatically takes only three lines of code. The first one establishes the starting point for the sprite's upper-left corner (all screen positions are in play field coordinates); in this case the sprite starts in the play field's upper-left corner at 10,10. The second tells the sprite how many pixels you want the sprite to move when the movement timer fires off. With a horizontal delta of 3 and a vertical delta of 2 the sprite will move from the upper-left corner of the play field down to the lower-right corner in a roughly 45-degree line. The third line tells the

sprite how many milliseconds it should delay between movement requests. Here the code requests that the sprite should move every 20 milliseconds.

```
sprite->SetStartingPosition(10, 10);
sprite->SetAutoMove(3, 2);
sprite->SetAutoMoveTime(20);
```

There are only two functions methods for moving the sprite directly. The first, `MoveTo`, repositions the sprite so that its upper-left corner is at the coordinates passed to the function. The other function, `Offset`, displaces the sprite from its current position by the pixel distances passed into the function. In other words, `MoveTo` is used when you need to set the sprite's absolute location and `Offset` when you need to move it in relative amounts. Both functions are independent of the sprite's movement timer. When you use either of these functions the sprite will be erased and redrawn on the next cycle through your game loop.

You can combine automatic movement with the other two movement functions. You might want a sprite to move automatically until a certain condition occurs, like hitting the edge of the screen, and then use one of the other functions to reposition the sprite so it can continue its automatic march across the screen.

SetStartingPosition

Let's look at the automatic methods of the sprite first, starting with setting the sprite's starting position, `SetStartingPosition`. As the name implies you should use this function to establish the sprite's starting location.

```
void  CSprite::SetStartingPosition(short h, short v)
{
      fCurrBnds.top = v;
      fCurrBnds.left = h;
      fCurrBnds.bottom = v + fCurrHeight;
```

```
        fCurrBnds.right = h + fCurrWidth;

        fEraseBnds = fCurrBnds;
        fPrevBnds = fCurrBnds;

}
```

The sprite's bounding box is adjusted to match the new position and then those bounds are copied to the erasure and previous position rectangles maintained by the sprite. This copying wipes out any history previously maintained by the sprite. So if the sprite was already in a play field it won't be erased correctly before being moved. If you wish to move a sprite that already has a position established in a play field, you'll want to use the `MoveTo` function. This function is to be used only for establishing the sprite's initial position within the play field.

SetAutoMove

Setting the number of pixels you want the sprite to move on each timer heartbeat is easy enough with a call to `SetAutoMove`. This function takes the number of pixels you want the sprite to be offset by on each cycle of the movement timer. Negative values represent movements to the left and up, and positive values move the sprite to the right and down, just like QuickDraw relative movements.

```
void  CSprite::SetAutoMove(short dh, short dv)
{
        fMoveHoriz = dh;
        fMoveVert = dv;
}
```

The function only assigns the deltas you pass to the fields maintained by the sprite. Notice that the sprite is not moved by a call to this function. The actual sprite movement doesn't take place until

the sprite's main movement function, `Move`, is called from the play field's `MoveSprites` function during your game loop.

SetAutoMoveTime

Establishing the frequency at which you want your sprite to move is done through `SetAutoMoveTime`. You need to pass the function the number of milliseconds delay you require between sprite movements. If you pass the flag `kASAP` as the delay amount you'll force the sprite to be moved as often as possible, which really means the sprite will be moved on every call to the play field's `MoveSprites` function. Which as far as your game is concerned is as often as possible.

If you don't want your sprite to move at all, pass in the enumerated value `kNoMovement`. With this delay value your sprite's delay is now, depending on how you look at it, either zero or an infinite delay. Either way the timer will never fire and your sprite's core movement function, `Move`, will never be called.

If you end up passing a real delay value and not one of the enumerated constants you'll force the sprite's movement timer to be reestablished using the delay you passed in.

```
void  CSprite::SetAutoMoveTime(long moveDelay)
{
      if(moveDelay == kASAP)
      {
            fTimerMoveFlag = TRUE;
      }
      else if(moveDelay <= kNoMovement)
      {
            // if the task is primed...
            if (IsTaskPrimed(&fMoveTimer))
            {
                  RmvTime((QElemPtr)&fMoveTimer);
                  InsTime((QElemPtr)&fMoveTimer);
            }

            // Don't let the sprite ever move
```

```
            fTimerMoveFlag = FALSE;
        }
            // Use the delay passed in
        else if (!IsTaskPrimed(&fMoveTimer))
        {
            // Force the sprite to allow movement.
            // This will force a reset of the timer
            // task on the next pass of the game loop
            fTimerMoveFlag = TRUE;
        }

        fMoveDelay = moveDelay;
}
```

Internally the function works by dispatching upon the delay passed in and setting up the sprite timer values appropriately. The first test is against `kASAP`. If the test is true the sprite's timer flag, `fTimerMoveFlag`, is set indicating to the world outside of the sprite that the sprite's timer has fired. Every time the play field inquires if enough time has passed between sprite movements it will be told yes, forcing the sprite to be moved as fast as the play field can manage.

Next the delay value is checked against the `kNoMovement` flag. If the delay indicates that the caller does want the sprite to freeze at its current location and stay there until told otherwise, the function first checks to see if the movement timer is currently active. If so, the function removes it and disables the timer with calls to the time manager functions `RmvTime` and `InsTime`. The first function removes the task from the time manager list of active tasks. The next line reinserts the function back into the same queue. The newly installed task will just take up space in the time manager's queue until a call to `PrimeTime` is performed on the task. The sprite's movement timer flag, `fTimerMoveFlag`, is then forced to false. With this flag set to false every inquiry performed by the play field to the sprite asking if the sprite needs to be moved will be rebuked, forcing the sprite to remain frozen at its current location.

If the caller did bother to pass in an actual delay amount, the function will check if a timer task is currently pending. If not, the `fTimerMoveFlag` is forced to true. This will force a reestablish-

ment of the timer with its new delay amount on the next pass through the game loop. If there is a timer task pending, nothing is done. The new delay value will be established after the current timer task is spent. This means that if you accidentally pass in a huge value for the delay amount, like say `MaxLong`, you'll have to wait 24 days before you can establish a new movement delay.

After handling all the combinations of possible delay values, the function finally assigns the new delay amount to the sprite's movement delay slot, `fMoveDelay`.

Movement Central

All of the sprite's automatic movement functions eventually get the sprite animating through its `Move` function. This function is movement central for all of the other movement functions. The other automatic functions only set variables maintained by the sprite that are used by the `Move` function.

```
void  CSprite::Move()
{
      if (fMoveHoriz || fMoveVert)
      {
            OffsetRect(&fCurrBnds, fMoveHoriz, fMoveVert);
            fDrawMe = TRUE;
      }

      if (fMoveDelay > kASAP)
      {
            // Reset timer
            fTimerMoveFlag = FALSE;

            PrimeTime((QElemPtr)&fMoveTimer, fMoveDelay);
      }
}
```

If either of the two automatic delta variables has a variable greater than zero, the sprite's bounding rectangle is offset by the amount stored in those variables. To make sure that the sprite will

be redrawn on the next animation cycle, the sprite flag, `fDrawMe`, which says to the world, "Hey, I need to be redrawn," is set to true.

Next the function check to see if the sprite's movement delay is a real delay value. If so, the timer is reset by first clearing the sprite's timer flag and then starting up the sprite's time manager task with a call to `PrimeTime`.

Direct Sprite Movement

To complement the automated ways of moving a sprite there are other sprite movement functions, `MoveTo` and `Offset`. Both functions move the sprite's bounding rectangle to a new on-screen position, effectively moving the sprite. And like the other sprite movement functions these two functions don't perform any blitting, instead setting the sprite's internal flags that reflect that the sprite needs to be blitted.

MoveTo

As previously mentioned, the `MoveTo` method of the sprite lets you move the sprite to an absolute location within the sprite's play field. On the next pass trough the game loop the sprite will be erased at its previous location and redrawn at the one specified by the arguments to `MoveTo`.

```
void  CSprite::MoveTo(short h, short v)
{
      if( h || v)
      {
            fCurrBnds.top = v;
            fCurrBnds.left = h;
            fCurrBnds.bottom = v + fCurrHeight;
            fCurrBnds.right = h + fCurrWidth;

            fDrawMe = TRUE;
      }
}
```

The code for `MoveTo` moves the sprite by reconstructing the sprite's bounding rectangle with its upper-left corner matching the horizontal and vertical positions passed in. After that the sprite drawing indicator flag is set to true so that on the next pass of the play field animation cycle the sprite can inform the play field that it needs to be redrawn.

One problem that can arise with `MoveTo` is if you decide to move the sprite in large chunks while the sprite is visible. Doing this will end up creating a blitting bounds that is the union of the sprite's previous location and its new location. If you moved the sprite from the lower-right corner up to the upper left you'd end up creating a blit from the working buffer to on-screen that is about the same size as the whole screen. Can you say *incredibly slow*? To avoid this particular situation you'd be better off hiding the sprite than moving it.

Offset

Rounding out the finalists in the sprite movement hit parade is the sprite function `Offset`. This nice little function offsets the sprite from its current position by the number of pixels specified by the horizontal and vertical deltas passed into the function.

```
void  CSprite::Offset(short dh, short dv)
{
      if (dh || dv)
      {
            OffsetRect(&fCurrBnds, dh, dv);
            fDrawMe = TRUE;
      }
}
```

Like the function `MoveTo` this function repositions the sprite's bounding rectangle to reflect the sprite's movement. And after the changes to the bounding rectangle, the sprite's drawing flag is set to reflect that the sprite's current on-screen position is stale and needs to be updated at the play field's earliest convenience.

Changing the Sprite's Cel

The other aspect of sprites controlled by a timer is the rate at which the sprite cycles through the cels owned by the sprite. Like sprite movement the sprite cels can be cycled automatically or manually. It's up to you.

SetCelCycleTime

A call to `SetCelCycleTime` tells the sprite how long it should wait before cycling to the next sprite cel. This method of the sprite is a mirror function of the sprite's `SetAutoMoveTime` function. It even takes the same parameters: the number of milliseconds of delay between changes or the enumerated constant `kASAP` to indicate that the sprite should cycle through the cels as fast as it possibly can, and the other constant `kNoMovement` to tell the sprite that you never want the sprite's cel to change.

```
void  CSprite::SetCelCycleTime(long cycleDelay)
{
      if(cycleDelay == kASAP)
      {
            fTimerCelFlag = TRUE;
      }
      else if(cycleDelay <= kNoMovement)
      {
                  // if the task is primed...
            if (IsTaskPrimed(&fCelTimer))
            {
                  RmvTime((QElemPtr)&fCelTimer);
                  InsTime((QElemPtr)&fCelTimer);
            }

            // Don't let the sprite ever move
            fTimerCelFlag = FALSE;
      }
             // Use the delay passed in
      else if (!IsTaskPrimed(&fCelTimer))
```

```
        {
                // Force the cel to change
                // This will force a reset of the timer
                // task on the next pass of the game loop
                fTimerCelFlag = TRUE;
        }

        fCelDelay = cycleDelay;
}
```

The function's implementation is also a mirror of `SetAutoMoveTime`. Depending on the value of the delay, the sprite's cel change flag `fTimerCelFlag` is either permanently set to true or permanently set to false and the timer task shut down. Or if you pass in an actual millisecond amount, the flag is set and the timer reset. Somewhere at the end of the function the code finally gets around to copying over the new delay amount to the sprite's new `fCelDelay` variable.

SetCurrentCel

You specify which cel of the sprite is to be used by the sprite by passing the cel's index in the sprite's cel list to the function `SetCurrentCel`. The indices start at one and rise to the number of cels. If you pass in an index less than one you'll get the first cel. Pass in an index larger than the number of cels and you'll get handed back the last cel in the cel list.

If you have established an automatic cel change with a call `SetCelCycleTime` this function serves to set the starting cel for the cel changing animation. After the proper delay the sprite will switch to the next cel in the list. When the sprite cel cycling reaches the last cel the sprite starts over at the beginning of the cel list.

On the other hand if you don't have the cel cycling timer running `SetCurrentCel` serves to tell the sprite to use the cel pointed at by the passed index. Without the timer the sprite will use the requested cel indefinitely.

GetCurrentCelIndex and GetCurrentCel

Use the sprite's `GetCurrentCelIndex` function to determine the index of the cel that the sprite is displaying. If you want access to the current cel instead of its index, use `GetCurrentCel` instead.

The `GetCurrentCelIndex` function is useful when you want to cycle through the cels manually.

```
sprite->SetCurrentCel( sprite->GetCurrentCelIndex() + 1);
```

Both `GetCurrentCelIndex` and `GetCurrentCel` are implemented as inline functions. And since their only code is a return statement the compiler should have no problem actually inlining these functions.

```
short GetCurrentCelIndex() { return fCelIndex; }
CSpriteCel *GetCurrentCel() { return fCurrentCel; }
```

Be aware that `GetCurrentCel` could potentially return a nil sprite cel object if the sprite doesn't go to the effort of building a couple of sprite cels. It could happen, so code accordingly.

ChangeCel

The function `ChangeCel`, like its sprite movement counterpart `Move`, is the core function used by the play field to cycle through the cels owned by each sprite. The play field calls the sprite's `ChangeCel` once each time through its event loop for each sprite that returns true to its `IsTimeToChangeCels` function.

```
void  CSprite::ChangeCel()
{

      fCelIndex++;
      if(fCelIndex > fCels->GetObjectCount())
            fCelIndex = 1;
```

```
        SetCurrentCel(fCelIndex);

        if (fCelDelay > kASAP)
        {
                // reset the timer
                fTimerCelFlag = FALSE;

                PrimeTime((QElemPtr)&fCelTimer, fCelDelay);
        }
}
```

The `ChangeCel` function bumps the sprite's current cel index by one, and if the new index is still within range the cel is changed with a call to `SetCurrentCel`. After the cel is changed the timer flag for cel cycling is reset and the cel timer reset through a call to the time manager's `PrimeTime` routine.

Sprite Visibility

You control the sprite's on-screen visibility with the sprite's twin complementary functions, `Hide` and `Show`. These two functions are about as self-explanatory as you can get. The only note, indicated by their implementation, is that the sprites don't change their visibility directly through these functions. The sprites won't change their on-screen state until the next go-through of the play field's animation loop.

These two functions come in handy when you would like to reposition a sprite without having to move it on-screen. Also they're awfully useful during collisions. After detecting a collision you can get rid of the collided with sprite with a quick call to `Hide`.

```
void  Hide()
{
        fVisible = FALSE;
        fDrawMe = TRUE;
        fEraseMe = TRUE;
}
```

```
void  Show()
{
      fVisible = TRUE;
      fDrawMe = TRUE;
      fEraseMe = FALSE;
}
```

Both functions work by toggling the appropriate sprite's visibility flags. To hide the sprite the visibility flag, `fVisible`, is set to false, telling the sprite to act like it's invisible. The drawing and erasing flags, `fDrawMe` and `fEraseMe`, are set to true so that on the next pass through the animation loop the now visible sprite will be erased from the screen.

Making a sprite visible on-screen is performed with a simple inversion of the sprite's `Hide` implementation. Flip the flags and on the next animation cycle the sprite will pop on-screen.

Sprite Miscellany

The base sprite class has only a couple of functions that fail to fall into any of the categories presented earlier. To avoid coming up with a potentially original category let's just throw these few functions under the sprite mélange banner and be done with it. Mélange. What a great word. Would make a great name for a new programming language.

"What did you program that incredible game in?"

"Mélange."

"Uhm, yeah, right."

It's better than Dylan and odds are that there won't be any off-key, own-press-believing folk singers dragging themselves out of impending obscurity to sue you.

Sprite IDs

When two sprites slam face-first into each other, you might want to know who ran into what. That's where the first two functions of our sprite's mélange (it's still a great word) come in handy.

I've referred to the sprite identification field as its dog tag for no other reason than that I liked the idea of programmers' having to check the sprite's dog tags after a fatal collision. Ignoring the overly macho name, a sprite's dog tag is simply a four-character identifier like the ones you use with the Mac's resource manager. You can tag the sprite with any value that fits within a long integer, but I like to use tags that I can read in the debugger. Tags like 'rock' and 'ship' are so much more readable at 3 A.M. than a tag like 2,345,235,678.

To assign a sprite's dog tag use the inline function `SetDogTag`.

```
void  SetDogTag(SpriteID id) { fDogTag = id; }
```

Retrieving a sprite's dog tag is done by the simple replacement of the *S* in the previous function's definition with a *G*, turning `SetDogTag` into `GetDogTag`.

```
SpriteID    GetDogTag() { return fDogTag; }
```

Like `SetDogTag`, GetDogTag is implemented as an inline declaration so that you won't feel tempted to make the `fDogTag` field visible to the public.

Movement Extents

Just like small children and programmers, sprites need to know their limits. In a sprite's case its limits are expressed as a rectangle that defines the extent that the sprite should be allowed to move. You establish a sprite's bounding limits with a call to the sprite's `SetSpriteExtent` method.

```
void  SetSpriteExtent(const Rect * extent)
{
      fMoveExtent = *extent;
}
```

Following suit with the sprite's dog tag functions, retrieving the sprite's cautionary boundaries is provided with a `Get` instead of a `Set`.

```
void  GetSpriteExtent(Rect * extent)
{
      *extent = fMoveExtent;
}
```

The default sprite implementation, again like small children and programmers, ignores the limits established by the boundary rectangle. These functions are provided as a convenience for future sprite classes you might concoct that have a need to know their place and how far they are allowed to wander within it.

10

Putting It All Together

Time to put all that sprite code to some use. A use, and maybe even a good one, would be a simple example game that ties in most of the pieces of the sprite class kit. And as fortune and preplanning would have it, that is exactly what this chapter does.

I thought a fitting first example would be to produce a version of the first video game, Pong. What could be a better example than coding up a tribute to the game that started it all. Somewhat like a rite of passage for the budding game programmer. Pong has all the essentials for a great first example: small number of objects on-screen, not a lot of complicated algorithms to get in the way, and little chance of any of Atari's lawyers showing up at my doorstep.

And to doubly insure that my entryway remains an attorney-free zone, let's name this copy of Pong "Ping." Did I say "copy"? Sorry, I meant homage. Let's name this homage to Pong "Ping."

Game Rules

For those of you who somehow missed the '70s here is a quick overview of how Pong plays out. The game has only three elements on-screen: two paddles and one ball. The game begins with the ball being fired at one of the paddles. The paddles can only move up and down. You want to move your paddle so that it deflects the path of the ball toward the other paddle. If you miss the ball the other players scores a point. First player to score eleven points wins. Amazingly, a copy of this simple game made it into almost every home in America. I personally believe the hypnotic simplicity of this game was directly responsible for Jimmy Carter's being elected and the success of Abba.

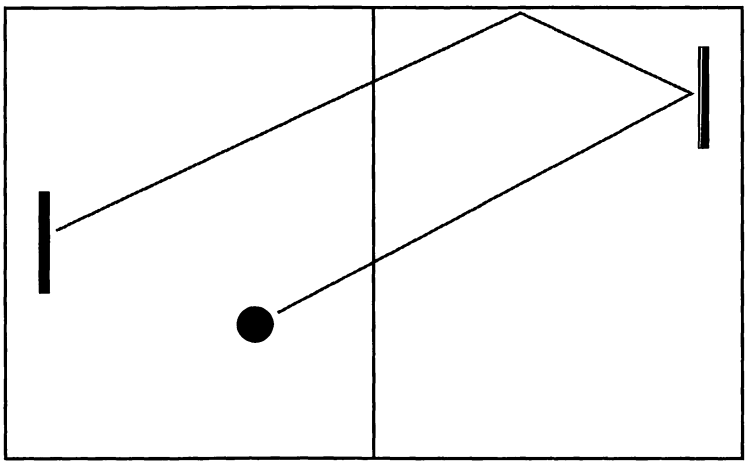


Figure 10-1. Pong

In Figure 10-1 the player controls the paddle on the left side of the screen with the Mac's mouse, with the computer controlling the

paddle on the right, leaving the ball controlled by some incredibly simplified rules of physics.

This example skips all those fluffy game details like scoring, menu bars, high scores, or anything else that would have you confuse it with anything other than what it really is, an excuse to try out all that sprite code. Fire up the game and you're instantly playing, first mouse click and the game quits. Can't get much simpler than that. Try it once or twice before heading on to the other sections of this chapter.

Ping's Code—Overview

To build the game Ping you'll need three sprites, one window, one play field, and a couple of sprite groups. Mix together with only a few pages of code and you end up with a cheap silicon souvenir of the seventies.

The game design breaks down into four problems:

- Tracking the mouse's movements with the player's paddle
- Making the computer's paddle play a reasonable game
- Collisions between the ball and paddles
- Bouncing the ball off the edges of the window

Bouncing Ball

Let's be original and handle the last problem first: how to get the darn ball to bounce off the edges of the window. First off the ball will have to be moving. Easy enough, use the sprite's `SetAutoMove` function with a predefined speed. With the ball moving you'll need a point for checking if the sprite has hit the bottom or top edges of the play field. Overriding the sprite's `Move` function provides a perfect vantage point for keeping the sprite in check.

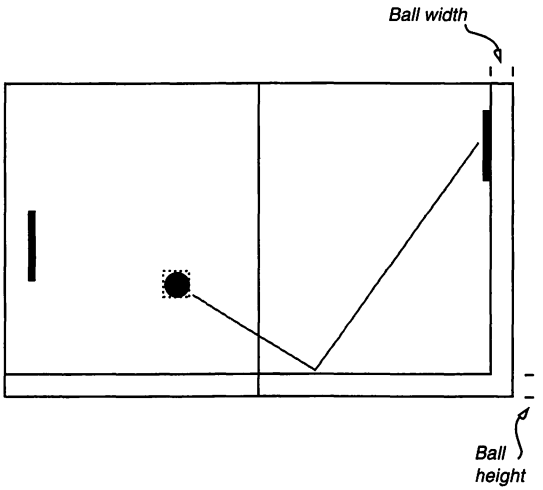


Figure 10-2. Perceived play field

The ball's movement will need to be checked against the bounds of the user-perceived play field. Which is not to be confused with the actual play field. The perceived play field for the ball is shown in Figure 10-2 by the dotted rectangle. This rectangle is the sprite's extent rectangle and is calculated by insetting the right and bottom edges of the play field by the width and height of the sprite. With this extent rectangle the `Move` method can use the sprite's top-left corner as the test case against all the extent's edges.

```
void  CBall::Move()
{
      // See if the ball has banged into
      // the walls or past a paddle
      if (fCurrBnds.top > fMoveExtent.bottom)
      {
            // Flip the sprite's direction
            fMoveVert = -fMoveVert;

            // Correct the sprite's position
            OffsetRect(&fCurrBnds, 0,
                  -(fCurrBnds.bottom - fMoveExtent.bottom));
      }
      else if (fCurrBnds.top <= fMoveExtent.top)
```

```
        {
                // Flip the sprite's direction
                fMoveVert = -fMoveVert;

                // Correct the sprite's position
                OffsetRect(&fCurrBnds, 0,
                                fMoveExtent.top - fCurrBnds.top);
        }

        // See if the ball made it past a paddle
        if(fCurrBnds.left < fMoveExtent.left)
        {
                // Made it past the player's paddle.
                // Reset the ball's position
                MoveTo(fMoveExtent.right + 40,
                                fMoveExtent.top - 40);
                SetAutoMove(-kBallDelta, -kBallDelta);
        }
        else if(fCurrBnds.left > fMoveExtent.right)
        {
                // Made it past the computer's paddle
                MoveTo(fMoveExtent.left - 40,
                                fMoveExtent.top - 40);
                SetAutoMove(kBallDelta, kBallDelta);
        }
        else
                inherited::Move();
}
```

During the `Move` function the ball's current position is checked against its extent. When the ball slams into the top or the bottom you'll just need to flip the direction of the sprite's vertical component of its automatic movement pair. A negative velocity—propelling the ball upward—will be inverted when the ball hits the top of the window, giving the ball a positive velocity. Ditto but in reverse for the bottom. While the flipping of the sprite's velocity leaves a little to be desired in the physically accurate department, it looks and acts accurate enough for our game's needs.

Once the ball has been checked against the top and bottom extents, it's time to see if the ball has made it past either of the paddles with a check of the ball's left edge. If the ball has made it past

one of the paddles the function resets for the next serve of the ball. If the ball scored for the player, the next serve will be toward the computer. Reverse that if the computer sneaked one past the player. In a real game this is the point you would change the scoreboard and check to see if the game has been won. But this is only an example so we do squat besides changing the serving direction.

Mouse Tracking

Getting the player's paddle to echo the mouse's movements is another task custom-made for overriding the sprite `Move` method. Instead of giving the player's paddle an automatic velocity you'll use the `Move` function to look at the mouse's current vertical position—with a call to `GetMouse`—and then position the paddle to match with a call to the sprite's `MoveTo` method. Since the paddles only move up and down, the mouse's horizontal position is completely ignored.

```
void  CPlayerPaddle::Move()
{
      Point mouse;

      GetMouse(&mouse);

      // If the mouse is trying to move the paddle
      // past its limit then pin the paddle within the
      // sprite's extent
      if (mouse.v < fMoveExtent.top)
            MoveTo(fCurrBnds.left, fMoveExtent.top);
      else if (mouse.v > fMoveExtent.bottom)
            MoveTo(fCurrBnds.left, fMoveExtent.bottom);
      else
            // Move the paddle to its new location
            MoveTo(fCurrBnds.left, mouse.v);

      // Make sure we get redrawn
      fDrawMe = TRUE;
      inherited::Move();
}
```

The only matter that'll you'll have to worry about is whether your window is smaller than the screen it resides on. Which will be true for this example. In this case `GetMouse` can give you coordinates that would fling your paddle right off the play field. So after adjusting the paddle to match the mouse you'll want to pin the paddle's sprite to the visible portion of the window. After adjusting the paddle make sure you indicate that you want the sprite redrawn on the next cycle of the game loop.

The illusion of the player's paddle moving smoothly is attained by the assumption that the mouse hasn't had a chance to move too much between frames of animation. This, of course, depends on sampling the mouse position frequently as the player moves it. Too much time between mouse-position snapshots and the paddle will appear to jump around the screen instead of smoothly sliding into position.

Paddle AI

To make Ping a game worth playing for three minutes (that's about all it's worth) the computer needs to be able to hit the ball with its paddle. More accurately, you want the computer to have to move its paddle in the same manner as the human player. You can't just plop the computer's paddle right in front of the ball at the last minute. That would be cheating. You need for the computer to track the ball with its paddle, smoothly bringing the paddle from one end of the screen to the other so it can get a good whack at the ball.

```
void  CComputerPaddle::Move()
{
      Rect  ballBnds;

      // The computer's paddle moves by tracking the ball's
      // position. If the ball is moving towards the
      // computer's paddle and it has passed center ice
      // start the paddle moving up or down.

      gBall->GetPosition(&ballBnds);
```

```
        if( ballBnds.right > gHalfWindowSize &&
                              gBall->GetAutoMoveHorz() > 0)
        {
              // Determine which direction the paddle should
              // be moving. The function MissFactor() gives a
              // random error term that allows the computer's
              // paddle to occasionally miss.

              if (ballBnds.top + MissFactor() < fCurrBnds.top)
                    fMoveHoriz = -kComputerPaddleSpeed;
              else if (ballBnds.bottom -
                          MissFactor() > fCurrBnds.bottom)
                    fMoveHoriz = kComputerPaddleSpeed;
              else
                    fMoveHoriz = 0;
        }
        else
              fMoveHoriz = 0;

        inherited::Move();
}
```

Since the computer will be moving the paddle it makes sense to override the computer paddle's `Move` function for our ball tracking. Within this function you'll get the ball's position and compare it to the computer's paddle. Based on the difference between the ball's position and the computer's paddle, the `Move` function will adjust the paddle's vertical velocity direction so that the paddle moves in the direction of the ball.

You'll notice that the function doesn't start tracking the ball position unless it is traveling toward the computer's paddle. It would look very mechanical for the paddle to be tracking the ball when it doesn't have to. As another realistic touch, the function delays tracking a ball heading toward it until the ball has passed the play field center line. This small addition stops the computer from jittering its paddle like it's hopped up on Jolt cola.

The final attempt at simulating a human foe is giving the computer a way to fail. On each look at the ball's position the function will make a call to the ball's private `MissFactor` function. This function randomly returns a factor that is applied to the paddle's

top position. This fudge factor adds some random slop to the paddle's tracking, allowing it to occasionally miss the ball.

```
short CComputerPaddle::MissFactor()
{
      return Random() % 2 == 0 ? 8 : 0;
}
```

The best way to understand these simplistic attempts at a realistic opponent is to comment them out and watch the game's reaction. You'll then notice that this simple feedback loop is a reasonable compromise between a passable implementation and getting a degree in artificial intelligence.

Volleying

Our only remaining challenge is making sure that the ball bounces off the paddles instead of traveling right through them. Testing whether two sprites have touched each other? Sounds like collision detection to me. But what to test, the paddles or the ball? It's an existential problem. Is the ball colliding with the paddle or is the paddle colliding with the ball? I'm a ball-hitting-the-paddle kind of guy. I also always see the glass as half full and think that a tree does make a sound and you can fit only fifty-five angels on a pin, but only if they know each other rather well.

To provide for the thrill of Ping volleys let's override the ball's `Collision` function. With the paddles being the only other sprites around for miles, you can be pretty sure when the ball's `Collision` method is called you have had a legitimate head-on with one of the paddles. Which one is irrelevant.

```
void  CBall::Collide(CSprite * source)
{
      short paddleSpeed;

      // Flip the ball's direction
      fMoveHoriz = -fMoveHoriz;
```

```
        // Use the paddle's momentum to change the ball's
        // relection angle and speed

        // Scale the paddle's delta to calculate
        // a psuedo speed for the paddle
        paddleSpeed = (source->fCurrBnds.top -
                                    source->fPrevBnds.top) / 4;
        if(paddleSpeed && paddleSpeed < 4)
              fMoveVert += paddleSpeed;

        Offset(fMoveHoriz, 0);
}
```

When the ball hits the paddle, and not the other way around, the ball's horizontal direction is reversed by inverting the ball's horizontal velocity. After giving the ball a 180, the `Collision` function tries to impart a little variation in the ball's vertical velocity. This change in velocity is based on the speed the paddle was moving at the time of collision. The speed of the paddle is faked by subtracting the paddle's previous position from its current position. This difference, or *faux* speed, is then scaled down by a factor of four, and if the scaled value is under the arbitrary threshold (in this case I again chose four) the scaled speed factor is combined with the ball's vertical velocity component.

The three sprites' class definitions are pretty simple to determine from the five member functions provided.

```
class CBall : public CSprite {
public:
      virtual     void  Collide(CSprite * source);
      virtual     void  Move();
};

class CPlayerPaddle : public CSprite {
public:
      virtual     void  Move();
};

class CComputerPaddle : public CSprite {
public:

      virtual     void  Move();
```

```
private:
      short MissFactor();
};
```

With these three Ping sprites fully defined let's move on to the fun part—stitching these sprites into a play field and finally into a game. A simple game, but a game nonetheless.

Ping's Code—Implementation

With a cache of Ping custom-designed sprites you now have the parts for building this fun little exercise in Newtonian mechanics. Though a game of Pong based on quantum mechanics would be a blast to play. But would the ball travel as a particle or a wave? I guess you wouldn't know until you ran the game.

Main, Where It Always Begins

The `main` for Ping does all the good things a well-behaved `main` function should. First let's fire up the Mac with a call to our handy-dandy utility function `InitMacApplication`. In this example the memory needs are simple so only two master pointer blocks and a overly generous stack of fifty kilobytes are generated.

```
void main()
{
        // fire up all the mac managers.
      InitMacApplication(2, 50 * 1024);

      // Install error handler
      SetSLVGErrorHandler(PingErrorHandler, 0L);

      // Create the play field and sprites
      BuildPingParts();

      // Play game
GameLoop();
```

```
        // Throw away the play field and sprites
            (gPingField);

          // Clean up any stray mouse clicks or key presses
          FlushEvents(everyEvent, 0);
}
```

Before launching into the game-part building code, `main` establishes the error handler for the sprite library used for this example. As you can see the error handling is incredibly robust. The kind you would expect in a *Dorf on Mac Programming* videotape.

```
void PingErrorHandler(OSErr error, long refcon)
{
      DebugStr("\pFatal Error. See Ya!");
      ExitToShell();
}
```

After installing the error handler the game parts are built through a call to `BuildPingParts`. This function will construct Ping's window, play field, and any needed sprites.

With the window and play field up and walking, the real action can finally begin. The function `GameLoop` launches into—you guessed it—Ping's game loop. The program will live in this loop until the user clicks the mouse button or hits the command option escape key combination. Either way the loop will exit.

If the player was kind enough to exit the game loop in the proper manner, control will return to `main` where it can clean up before hitting its final brace. Before that brace we get overly fastidious and throw away Ping's play field, disposing of the play field and all of its associated sprites and offscreens. This of course isn't necessary; quitting the game will nuke the entire heap of our sprites, offscreens, and anything else that was left lying around.

As a final nicety to the user before quitting, `main` flushes any events lying around in the event queue. This prevents the user who banged on the mouse like it was a stuck telegraph key from getting those extra mouse clicks sprinkled within the program that become active after Ping quits.

Building Them Sprites

Building the parts used for Ping is the responsibility of `BuildPingParts`. It constructs all the necessary game elements: window (got to have a window), sprites, play field, sprite groups. And ties all these necessary elements together. Along the way the function also caches most of these elements into global variables for easy retrieval at a later day.

```
void BuildPingParts()
{
      Rect                windowBnds;
      CSpriteGroup *      paddleGroup;
      CSpriteGroup *      ballGroup;

      // Create Window
      gPingWindow = GetNewCWindow (kPingWindowID, nil,
                                        (WindowPtr)-1L);
      windowBnds = gPingWindow->portRect;
      gHalfWindowSize =
            (windowBnds.right - windowBnds.left) / 2;

      // Create play field for game
      gPingPlayfield = new CPlayField( gPingWindow,
                                          &windowBnds);
```

Before we build any of Ping's animation elements, we need to erect a window on the screen to hold those elements. After the window has been reincarnated from its resource template description, half of its width is measured and stashed away in the global `gHalfWindowSize` for use with the computer paddle's ball-tracking code. With a Windex-clean window on-screen a play field is then constructed that takes up all the visible space within the window. The global `gPingPlayfield` will hold the reference to this new play field.

```
// Create the play field's sprite groups;
paddleGroup = new CSpriteGroup;
ballGroup = new CSpriteGroup;

// Create ball sprite
gBall = new CBall (kBallTemplateID);
```

```
gBall->SetAutoMove(-kBallsSpeed, kBallsSpeed);

// Create player's paddle sprite
gPlayerPaddle = new CPlayerPaddle (kPlayerPaddleID);

// Create computer's paddle sprite
gComputerPaddle = new CComputerPaddle
                                (kComputerPaddleID);
gComputerPaddle->SetAutoMove(0, kComputerPaddleSpeed);
```

Before creating Ping's sprites, we need a couple of sprite groups to hold those sprites. One group will hold the paddle sprites, the other the ball. I guess in that group's case the term *sprite group* is a misnomer. A whole group that holds only one lonely sprite. Seems kind of cruel. Oh well, game programming is cruel.

With a cruelly created sprite group just itching to get its hands on some new sprites, `BuildPingParts` does the right thing and creates the three sprites for the game. All three are constructed through sprite templates, avoiding a whole bunch of boring initialization code. Speaking of boring initialization code, here's some now that establishes the auto-movement rates of the ball and the computer's paddle. Remember that the player's paddle is moved by tracking the mouse and doesn't need or want any initial velocities.

```
// Fix up the sprite's movement extents
Rect  paddleExtent = windowBnds;

// Assumes that the two paddles are the same height
paddleExtent.bottom -= gPlayerPaddle->GetHeight();
gPlayerPaddle->SetSpriteExtent(&paddleExtent);
gComputerPaddle->SetSpriteExtent(&paddleExtent);

Rect  ballExtent = windowBnds;

// Suck in the the ball's extent at
// the right,left and bottom edges
ballExtent.bottom -= gBall->GetHeight();
ballExtent.right -= gBall->GetWidth() +
                                    kPlayfieldMargin;
ballExtent.left += gBall->GetWidth() +
                                    kPlayfieldMargin;
gBall->SetSpriteExtent(&ballExtent);
```

After establishing the sprite's initial velocities `BuildPingParts` goes the extra mile and calculates the movement extent rectangles for each sprite. The two paddle sprites are able to use the same extent bounds as long as they're the same height. If you decide to give the computer or the player a handicap by shortening one of the paddles, make sure you account for the new sizes in the paddle's extent calculation code. Luckily there is only one ball involved so its extent is adjusted for its height and width and then handed to the ball sprite. It'll know what to do with it.

```
    // Add the sprites to their proper groups
    paddleGroup->AddSprite(gPlayerPaddle);
    paddleGroup->AddSprite(gComputerPaddle);

    ballGroup->AddSprite(gBall);

    // Add the groups to the playfield
    gPingPlayfield->AddGroup(paddleGroup);
    gPingPlayfield->AddGroup(ballGroup);

    gPingPlayfield->HandlePlayFieldUpdate(&windowBnds);
}
```

The sprites are assigned to their proper sprite groups, and these groups are then placed in the care of the game's play field. Before returning to the caller `BuildPingParts` makes sure the window is correctly showing the beginning of the game with a call to the play field update method `HandlePlayFieldUpdate`.

Play That Game

Actual game play is accomplished from the `GameLoop` function. Here the program is held in a loop until the mouse button is detected as being held down. During that potentially endless loop the game executes the three necessary steps for getting the sprites to do their thing. All other needed sprite handling will be performed by the sprites themselves. After performing the requisite sprightly duties required of a proper game loop, the loop attempts to behave

like a proper Mac citizen and gives some processing time to the Operating System with a call to Mac's `SystemTask` function. Failure to do this could cause some of your background programs to start skipping a few beats. Not a problem if you're downloading a fresh batch of "art" from alt.binaries.pictures.your.mom.would.not.want.you.to.be.looking.at, but a major problem if you're experimenting with that new Mac heart monitor.

```
void GameLoop()
{
      while(!Button())
      {
            gPingField->MoveSprites();
            gPingField->CheckForCollisions();
            gPingField->ShowNextFrame();
            SystemTask();
      }
}
```

In a real game—wait, this is a real game—in a good game you would have the break condition tied to actual elements of the game, like the number of lives and the pause key. Future game loops will cover this.

Redecorating

With a fully functioning game example under wraps it's now time to show off the advantages of the sprite library. In the folder next to the Ping project is the alternative Ping project, Wacky Ping, which is based on the same source as the original Ping. All that's been changed are the sprite templates. You can think of this as a beauty makeover for Ping. Just like on Jenny Jones. We started out with the Plain Jane or Plain Joe look of our original Ping and performed a pixel makeover.

The two paddles are now constructed out of some really nice scans of fine polished exotic wood. Brushed steel seemed too cold for a friendly game of Ping. The ball sprite has been given the appearance of a real tennis ball, complete with a few additional sprite cels to give it that slowly-rotating-ball-in-zero-gravity-with-absolutely-no-friction look.

Now the game not only looks better—well, if not better at least different—and with its new look it now takes up twice as much disk and memory space as the "before" version. Bigger is better. Right?

Experiment

Now it's your turn. Take this simple example and do something goofy with it. Maybe you can build a cross between Mortal Kombat and Pong. Call it Primal Pong. You could have the paddles spew blood every time they return a volley. The ball could be one of those polished chrome Phantasm Swiss Army Knife death balls. I hate that ball. I still break out in a cold sweat whenever I'm in a room with some ball bearings.

11

Sound

Sound is the second most essential element of any game, the first being graphics. If you could make a game that used only sound, then sound would obviously be the most important element of that game, but for now I'm sticking with my original assertion. You can create a game without sound just like you can have sex without moaning, pizza without pepperoni, and Pink Floyd without Roger Waters, but why on earth would you want to? Sound is the essential icing for the great game recipe. The graphics of a game hold your interest, the design of the game play brings the players back for more, but it is sound that provides the total immersive experience that a great game gives you.

This chapter will help you on that road to the total Mac game experience. So drop some Hendrix into your CD player, crank the volume up to eleven, and start reading.

What Is Sound?

Time for an eighth-grade science review of sound. Put a CD on your stereo that has a good beat and that you can dance to. I would suggest some Parliament. George Clinton (no relation to Bill that I'm aware of) is great for science experiments. Remove the cover from one of your speakers and find the big speaker cone that is beating in a funkadelic fashion. Lightly put your hand on the speaker cone. On second thought, maybe you shouldn't perform this experiment on your own stereo. Try your local electronics megastore instead. They'll have much larger speakers and private little rooms with lots expensive speakers for you to experiment with. Remember to bring your own CD or you might be forced to listen to Dan Fogerty. No matter whose speaker cone you're destroying, your fingertips will feel the pulsating rhythm (unless you're listening to Dan) of the speaker cone as it moves back and forth in time to the beat. The motion of the speaker cone is producing the deep bass rhythms you are hearing. The higher-pitched sound is the store's manager. Ignore him for now as we'll get to pitch and frequency a little later in the chapter.

As the speaker cone vibrates back and forth in time to the music, it is pushing and pulling at the air surrounding the surface of the speaker cone. Air is composed of (use your best Carl "Humorless" Sagan here) billions and billions of tiny molecules. So when the speaker cone pushes forward, billions of those air molecules next to the surface of the cone are crammed together. These molecules react to the cramming by pushing against the molecules in front of them, and they then start pushing against their neighbors until a veritable riot of pushing molecules ensues. This riot sets a sound compression wave on its journey to your ear.

The single compression wave produced by the forward movement of the speaker cone will not be detected by your ear as sound.

As the cone continues to wiggle back and forth, each forward movement generates another compression wave moving toward your ear. Between each pair of these waves is an area of relative peace and calm where the air molecules have been spread apart more than they usually like. These moments of peace match the movement of the speaker cone inward. And it is the combinations of these compression waves and bits of silence that make up what your ear hears as sound. You can take your fingers off the speaker now.

Measuring Sound

Before you can talk about sound—"talk about sound," isn't that redundant?—you need to get the important sound terminology down.

Amplitude

If you were to put the output of your stereo into an oscilloscope you would see your stereo generating a picture of the sound like the one in Figure 11-1. Well, probably not like that unless you were listening to the emergency broadcasting signal.

Figure 11-1 represents the sound as a waveform display, what you would see with the oscilloscope coming out of your stereo, with time represented by the horizontal axis and the strength of the sound using the vertical axis. The vertical axis is what we're interested in at the moment—the strength of the sound signal given by the amplitude of the wave. The amplitude of the signal is what you are adjusting when you turn up the volume of your stereo. Some stereos even name that control the amplitude knob. No matter what it is called, the amplitude reflects the loudness of the sound signal.

The loudness of the signal is strictly a function of the height, or amplitude, of the waveforms that make up the sound. In Figure 11-2 the sound on the right is louder than the sound on the left. Though both are pretty boring sounds.

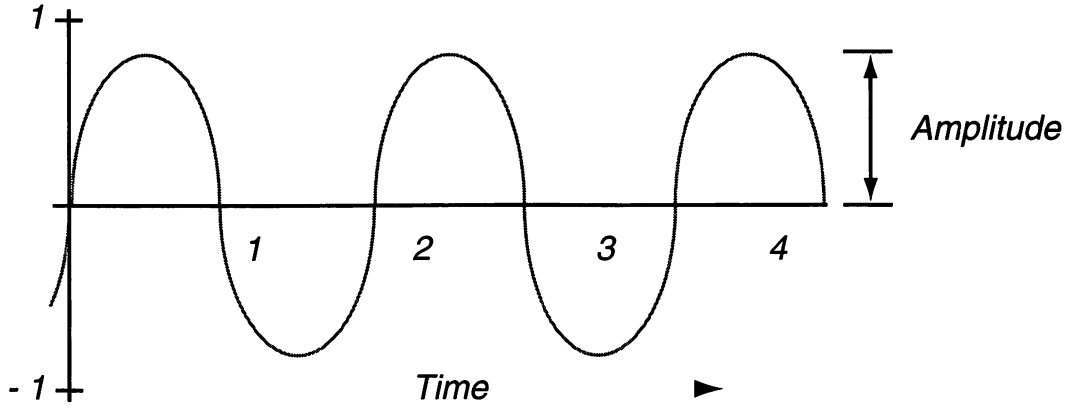


Figure 11-1. Sound as waveform

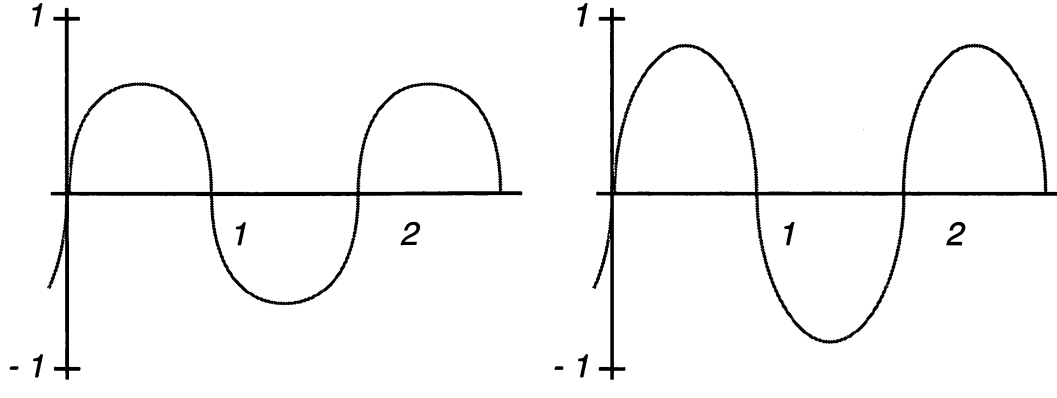


Figure 11-2. Right side is louder

The loudness of a sound is measured in bels. Named after Alexander Graham Bell (why they dropped the second *l* is a mystery to me). Actually one bel is too large to measure sound, so its smaller metric cousin the decibel (abbreviated dB, pronounced *dee-bee*) is used instead. The bel is a logarithmic scale of measurement. To increase the loudness of a signal by one whole bel, or ten decibels, you would have to increase the power of the signal ten times.

The human ear can only detect changes in the loudness of a signal of about one decibel. Below that the increase in power is indistinguishable. By increasing a signal's loudness by one decibel you are increasing the sound's strength by about 26 percent. The impact on the human ear means that while it isn't all that sensitive to incremental changes in volume, it has the ability to cover a large dynamic range of sound levels.

Your ears can hear a whisper across a room with a signal strength of 20 dB. They easily manage a normal conversation of about 40 dB. And they tolerate traffic noise at a busy intersection measuring in at 70 dB. You can even manage to be next to a small explosion that propels violent shock waves of 120 dB without going permanently deaf. Anything above 120 decibels becomes painful. Which would place a good Nine Inch Nails concert at 125 dB and worth every one.

What's the Frequency, Kenneth?

Measuring the amplitude of a sound will only tell you how loud the sound is. To get a mathematical representation of what the sound actually sounds like you need to examine the frequency of the sound wave.

To measure the frequency of a sound you first need to know what the period of the wave is. The period of a waveform is the distance the waveform traverses to complete one full cycle, or you can measure it (see Figure 11-3) as the distance from one peak of the wave to the next.

The frequency of a sound is then expressed as the number of periods or cycles the sound goes through in one second. The number

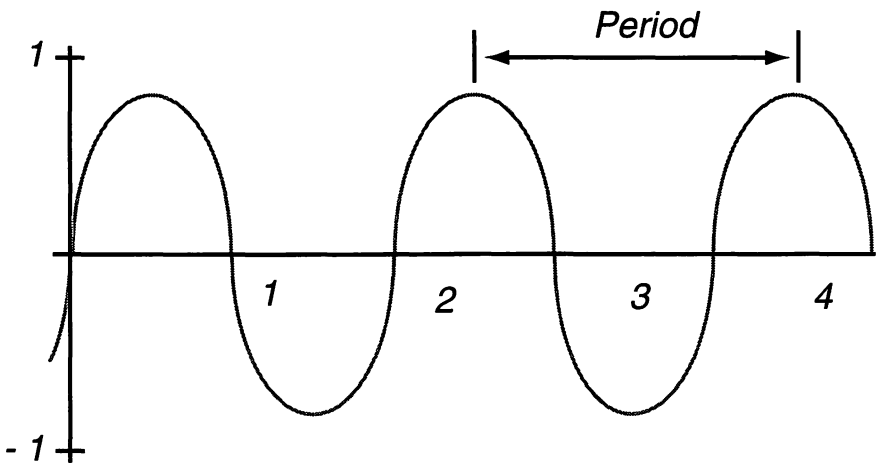


Figure 11-3. Measuring frequency

of cycles per second is usually expressed in units called Hertz (Hz), named after Heinrich Hertz, who after coining this term went on to make his fortune renting carriages to tourists visiting Hamburg.

Depending on how close to the concert speakers you were during your youth, you may be able to hear sounds with frequencies as low as a gut-vibrating 20 Hz to the high of a screeching 20 kilohertz (kHz). Pretty good range; not as good as my Labrador retriever, Bob, who can hear you thinking about food, but pretty good.

Digital Sound

All of the sounds your ears hear are analog sounds. Even if they are generated from digital sources, like your Mac, the sounds end up as analog waveforms bouncing off your eardrum. Problem is, your Mac isn't an analog computer. It can't store a sound as a continuous analog function. The best it can do is store a digital approximation of an analog sound.

For sound to come out of its speaker, the Mac has to convert the discrete digital representation of the sound it has stored into a continuous analog signal that your ears can understand. Somewhere in your Mac is a piece of silicon that does just that; it converts the digital sound in the Mac to analog impulses that drive the speaker. This nifty piece of circuitry is called a digital-to-analog converter or DAC for short. You have another DAC converting the digital frame buffer of your Mac into an analog signal that your monitor can display.

That explains how the Mac plays digital sounds, but how does it create them? Glad you asked.

Sampling

An analog sound source is converted to digital by first being converted to an electrical signal; that's what a microphone does. Then that signal is converted into its digital counterpart when the analog signal is sampled. Sampling is the process of capturing the signal's voltage, which equals the signal's amplitude, as a stream of digital samples, or numbers. Each of the numbers in this stream is a snapshot of the original signal's voltage at that specific moment in time. The hardware that produces this stream is referred to as an audio digitizer. Feed the captured numeric stream from a digitizer back out through the Mac's DAC and speaker combination and you will hear a signal that approximates the original.

And it is this approximation part that makes digital sound tricky. You want the digital sound to be as nearly accurate as the original as possible. The two factors that determine the accuracy of your digital copy are the sampling rate used to sample the original signal and the resolution that those samples are stored at.

Sampling Rate

The more frequently you can sample a sound source, the more accurate the digital representation will be. In Figure 11-4 a low sampling rate has produced a stream of samples that would not very accurately represent the original source signal.

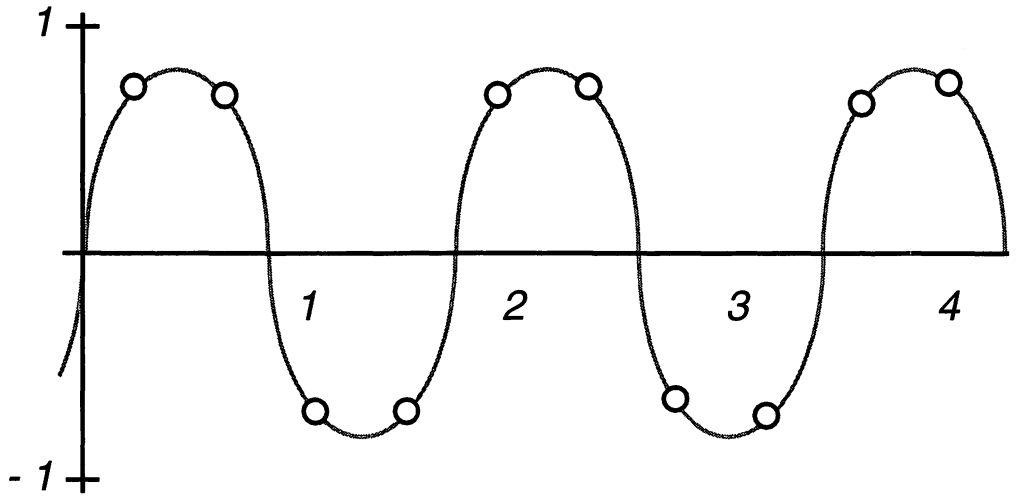


Figure 11-4. Low sampling rate

By increasing the sampling rate, or sampling frequency, as shown in Figure 11-5, you get a digital representation that can faithfully reproduce the original source signal. The rate or frequency that the sound is sampled at is measured in hertz. Just like the frequency of the analog source, isn't that convenient. The higher the frequency, the better the sound reproduction. For computers, common low-quality sampling rates are 7 and 11 kHz. These rates will give you about the same quality of playback as your telephone does. Good enough for a `SysBeep` but not much else. A medium-quality sampling rate is 22 kHz. This is the natural sampling rate of the Mac, and though you wouldn't want to listen to any music that you care about at this rate, it's good enough for most games. Commercial-quality audio is sampled at audio CD rates, 44.1 kHz, and digital audio tape rates of 48 kHz. Both provide sampling rates frequent enough to capture any source frequency that your ears could detect.

In a perfect world your sampling rate would be continuous, giving you an infinite number of samples for any slice of sample time. Of course you would need an infinite amount of memory and

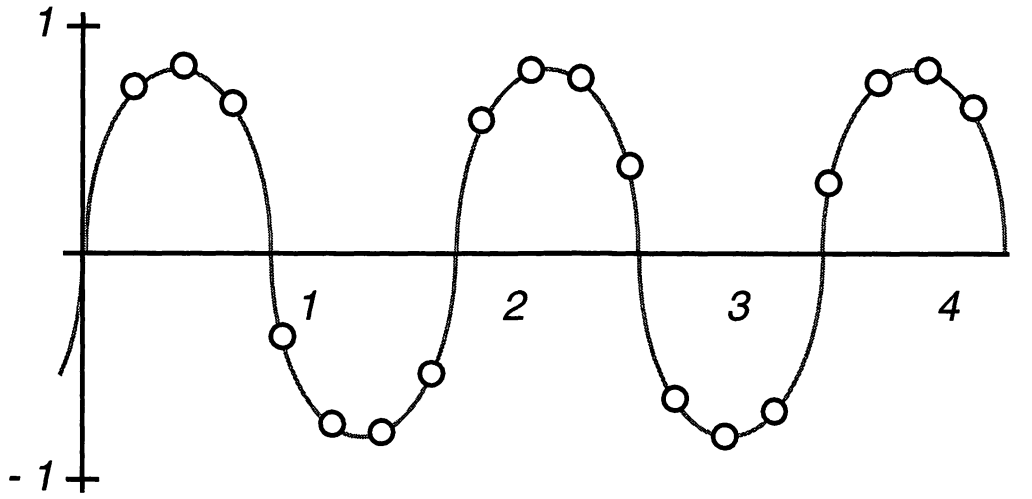


Figure 11-5. Higher sampling rate

disk space to store and play back this perfect sound. Without infinite storage you need to determine the proper sampling rate for a signal that will accurately represent that signal upon playback.

So what sampling rate will adequately represent the original signal? I don't know but a smart guy named H. Nyquist did. Nyquist derived that the minimum sampling rate is twice that of the highest frequency that the source signal contains. So if your source signal's highest possible frequency is 22.05 kHz (the highest frequency most people can perceive), then the minimum sampling rate would be twice that or 44.1 kHz. This calculated rate is known as the Nyquist rate.

If you decide to ignore Mr. Nyquist in order to save a few bytes you'll end up with a sample stream that cannot accurately represent the original source. In fact, the samples will sound like an altogether different sound. You will have created a clone or an alias of a lower-frequency sound. Having this happen is known as *aliasing*. The technique of trying to interpolate the needed extra information between the infrequent samples is known as antialiasing.

Sample Resolution

Sample frequency is only one part of the sampling Oreo cookie. The cream for that cookie is made from the sampling resolution. Each sample point must be contained in a finite-sized number of bits, which means that the voltage of the sound has to be scaled or quantized into that number of bits. If you use the Mac's typical sample size of 8 bits you'll have to map the strength of the source signal at each sample point into only 255 different values. You can think of the sample resolution as being the error term for each sample. In Figure 11-6 the quantizing error has been exaggerated to show the effects that 8-bit samples could have on a digitized signal.

The same signal sampled at the same frequency but with 16-bit samples would provide signal samples with almost no perceptible level of error (see Figure 11-7). Remember, it's not the number of bits that is important but the numeric range that those bits provide. With 16 bits the sample range has been extended from 255 different

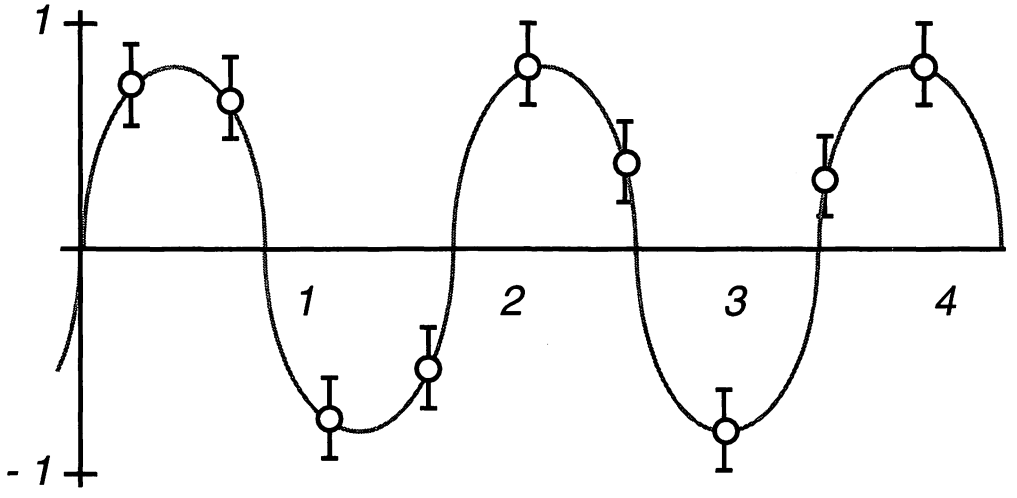


Figure 11-6. 8-bit sample size

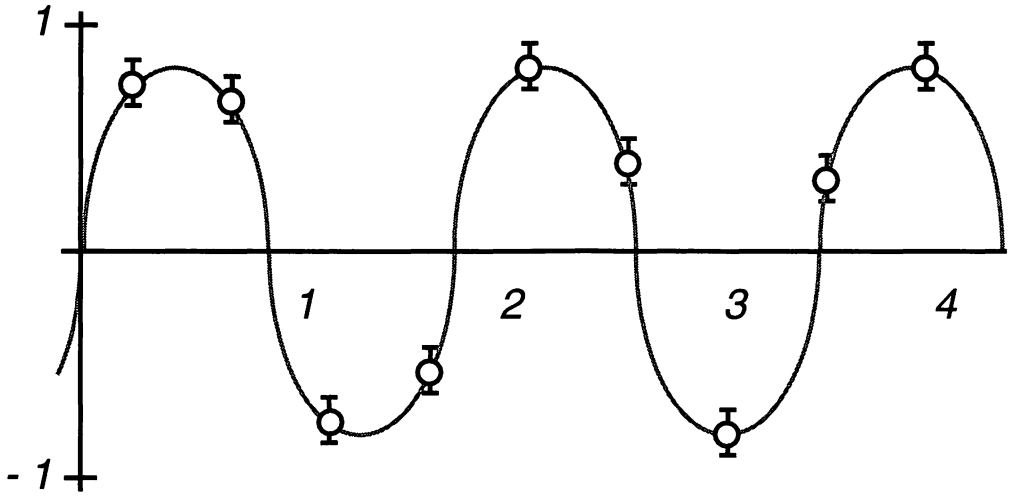


Figure 11-7. 16-bit sample size

voltage representations to a whopping 65,535. So by doubling the number of bits the sample's potential range has increased 256 times.

Why You Care

Why would you care all about this digital audio trivia? One simple reason is that sound quality equals space used—used by the memory that your game requires and the space that your game files take up on disk.

Since for reasonable playback on the Mac you must have the sounds you want to play resident in memory before you try to play them, the amount of memory they use is important. Digitized sounds can easily end up using more memory than the offscreen buffers used by the sprite engine.

Disk space is really only a problem in delivery. The larger your game, the more disks it takes to ship on, which can drastically affect

the shipping cost of goods and thus your profits. Or in the case of shareware the larger the sounds, the longer your players will have to spend on-line downloading your game. Stuff your game with big enough sounds and you could end up limiting your audience to those with super-fast modems. Though with a compressed installer the disk problem isn't as big an issue as the sounds chewing up run-time memory. Luckily, sounds compress rather well, usually yielding a 30- to 50-percent on-disk space savings. Regrettably, decompression takes time and can't be used at run time.

So the trick is to figure out what is a reasonable sample rate and size for your game that will provide enough audio fidelity and yet not eat up all your disk and memory space. On the Mac this isn't much of a trick. Most games use 8-bit samples at 22 kHz for all sounds. This is the natural rate and size that the Mac sound system expects to see. Try to feed it anything else and you'll slow down playback. Slow down playback and your sprites will start to creep across the screen. All the examples presented will be based around 8-bit samples with a 22 kHz frequency.

Sound on the Mac

Sound has always been an integral part of the Mac experience, from the startup gong to when it spoke those three famous words on its unveiling: "Hello, I'm Macintosh."

From that initial halting speech the Mac team has always made sure that the Mac was able to make noise. And with each new model the Mac's sound capabilities have increased. The software interface for these ever-increasing sound capabilities has always been through the Mac's Sound Manager.

Sound Manager: A Brief History

The Mac sound system has had an interesting past. From a simple driver that barely kept the speaker fed to the CD-quality sound playback systems of the current Mac lineup, it's been a case of

"You've come a long way, baby." So on with our history tour of the Sound Manager. Keep your hands in the vehicle at all times and please, no flash photography.

Sound Driver

The original Mac came with a sound driver. You wouldn't want to call it a Sound Manager as it barely managed play sounds. The original, phone-book edition of *Inside Mac* documented the sound driver's simple square wave synthesizer, which could produce Atari 2600-like buzzing sounds. It contained a four-tone synthesizer that allowed the programmer to set the waveform that would be used for each of the four channels. You could use any 8-bit sample you wanted as long as it fit into 255 bytes. And the coolest part of the Mac's sound driver was the sampled sound playback. With this part of the driver you could play back 22 kHz 8-bit samples that sounded really impressive. Unless your Mac was parked next to an Amiga.

You could play back sampled sounds with the sound driver but you had no built-in way of recording them. You'd have to drop $125 and buy yourself a MacNifty sound digitizer if you wanted to record annoying sounds for your Mac.

Sound Manager

Time marched on and the Mac II rolled out. Along with a 68020 and color displays, the Mac II had the first Apple sound chip, known as the Apple Sound Chip or the ASC by the technologically hip. With the chip came a whole redesign of the sound driver to support it. The new software interface to the sound hardware was so impressive that the name changed from the sound driver to the Sound Manager and a gained a huge chapter in the new *Inside Mac*, volume V. Trouble was, the only thing this new Sound Manager managed to do was lie. The documentation for the Sound Manager was not just slightly wrong, it outright lied. And where it did document features that actually existed, it was usually wrong.

Apple tried to fix up the Sound Manager with the release of System 6.0. Strike one. Not only were old bugs not fixed but they introduced new ones. Apple tried again with system 6.0.2. Strike two, a total whiff. Fixed more bugs and added a few more, though 6.0.2 did fix more bugs than it introduced. The preliminary new *Inside Mac* Sound Manager managed to lie about only a few things. Strike three, you're outta' here. Next batter.

Sound Manager Part II

The next Sound Manager to take the plate was the one contained in system 6.0.7, Sound Manager version 2.0. This Sound Manager came with a whole new chapter in the new *Inside Mac*, volume VI, a tome so large it has its own climate. This new manager documented how you could record sounds from your Mac without buying a third-party digitizer. How you could play sound files directly off the disk. How you could compress sounds in a format that the Mac could play back from other programs. How you could play multiple channels of sound. And for the game programmer, an approved way to handle double buffering. The best part about these features was that they actually existed. You could read about them and spend a few evenings coding and actually get sounds out of the speaker just as it was documented. This Sound Manager wasn't a home run, but it was a legitimate stand-up triple.

Sound Manager Part III

The pitch: low and inside. The swing. Whack! It's going . . . going . . . gone. With Sound Manager 3.0 Apple said good-bye to Mr. Spaulding and said hello to a whole feast of features. Backwards-compatible with Sound Manager 2.0, it also managed to squeeze in support for 16-bit, 44.1 kHz sound. A plug-in sound component architecture. Support for third-party sound boards. And best of all a two- to threefold increase in speed. This gain in speed allowed many game programmers to stop handling their own sound processing and use the Mac's. Which was the goal of the Sound Managers from day one.

Playing Sound

Enough history. Let's make some noise. This section will give you an overview of how to use the Sound Manager to produce sounds for your games. The information covered here will be used in the next section to construct a sound kit specific to games.

Sound Commands

Sound is generated by the Sound Manager through the use of sound commands. Every sound you play on the Mac is ultimately produced by directing sound commands at the Sound Manager. The commands supported by the Sound Manager are listed in the table.

```
enum {
      nullCmd                 = 0,
      quietCmd                = 3,
      flushCmd                = 4,
      reInitCmd               = 5,
      waitCmd                 = 10,
      pauseCmd                = 11,
      resumeCmd               = 12,
      callBackCmd             = 13,
      syncCmd                 = 14,
      availableCmd            = 24,
      versionCmd              = 25,
      totalLoadCmd            = 26,
      loadCmd                 = 27,
      freqDurationCmd         = 40,
      restCmd                 = 41,
      freqCmd                 = 42,
      ampCmd                  = 43,
      timbreCmd               = 44,
      getAmpCmd               = 45,
      volumeCmd               = 46,
      getVolumeCmd            = 47,
      waveTableCmd            = 60,
      phaseCmd                = 61
      soundCmd                = 80,
```

```
        bufferCmd                   = 81,
        rateCmd                     = 82,
        continueCmd                 = 83,
        doubleBufferCmd             = 84,
        getRateCmd                  = 85,
        rateMultiplierCmd           = 86,
        getRateMultiplierCmd        = 87,
        sizeCmd                     = 90,
        convertCmd                  = 91
};
```

When you pass one of these 33 commands to the Sound Manager you do so by first wrapping the command in a `SndCommand` structure. This structure has not only space for the command but space for two additional parameters that can go along with the command. Not all commands require parameters, but all communications with the Sound Manager must take place through this structure.

```
typedef struct {
        unsigned short    cmd;
        short             param1;
        long              param2;
} SndCommand;
```

We'll be using only a few of these commands in this chapter, so if you want the full dope on all of these commands drag out your copy of *Inside Mac: Sound*.

You can only send sound commands to the Sound Manager through three entry points. The first, `SndDoCommand`, executes the commands sent with it in FIFO order. This allows you to stack commands up faster than the hardware can process them without having to worrying about the Sound Manager dropping commands. Unless of course the command queue is full, in which case you'll be handed back a `queueFull` error. That is, you'll get that error back if you pass in True for the queue waiting parameter. Otherwise `SndDoCommand` will wait until an earlier posted command has been processed and removed from the queue in order to insert the fresh command you passed in.

```
OSErr SndDoCommand( SndChannelPtr chan,
                        const SndCommand *cmd,
                        Boolean noWait);
```

All commands handed to `SndDoCommand` are executed at interrupt level by the Sound Manager asynchronously to the main thread of your program.

The second method of executing sound commands is with a call to `SndDoImmediate`. As its name implies this function allows you to cut to the front of the command line and have your command executed without waiting for any commands in the queue to be processed. This function is useful for things like quickly shutting down sound playback.

```
OSErr SndDoImmediate( SndChannelPtr chan,
                        const SndCommand *cmd);
```

The third command-processing entry point is to package your commands in a `snd` resource that you then play back with a call to the Sound Manager's `SndPlay` command. The `snd` resource contains all the sound commands that you want to execute along with their parameters and any needed data.

```
OSErr SndPlay( SndChannelPtr chan,
                   SndListHandle sndHdl,
                   Boolean async);
```

The commands contained in the resource are executed from the start of the resource to the end. These commands will either be executed synchronously or asynchronously depending on the value of the Boolean passed as the `async` parameter. To refresh your memory, calling the `SndPlay` synchronously will force the caller of `SndPlay` to wait until the Sound Manager is done processing the commands contained within the `snd` resource. Calling `SndPlay` asynchronously lets the Sound Manager return control immediately to the calling function, with the Sound Manager executing the `snd` commands at interrupt time parallel to your program's execution.

The function SndControl was introduced as another method of sending commands to the Sound Manager with version 2.0. With Sound Manager 3.0 it is being phased out.

Sound Channels

As you probably noticed, all the methods of passing sound commands to the Sound Manager require a sound channel. Before you can generate any sounds on the Mac you have to open a path to the Sound Manager for your sound to travel. That path is a sound channel. Commands are sent to the Sound Manager through a sound channel, which manages the commands in a FIFO queue (unless the command was sent with SndDoImmediate). The Sound Manager finds all its commands within sound channels. Even calls to the Sound Manager that don't require you to pass in a sound channel will probably construct one internally.

```
typedef struct {
      struct SndChannel *nextChan;
      Ptr               firstMod; // Used internally
      SndCallBackUPP    callBack;
      long              userInfo; // Refcon
      long              wait;
      SndCommand        cmdInProgress;
      short             flags;
      short             qLength;
      short             qHead;
      short             qTail;
      SndCommand        queue[stdQLength];
} SndChannel ;
```

If you use the Sound Manager at a high level you'd probably never need to know what a sound channel is. But game programmers never get to use the high-level functions. In order to play sounds asynchronously you'll need to get down and dirty with the sound channels.

The Easy Way

The easiest way to play a sound from the Mac is to copy the sound you want to play into the system. Select that sound as the alert sound from the sound control panel. Call `SysBeep`. Bingo, your sound will beep. Easy, but not very practical.

The second-easiest way is to have your sound stored as a `snd` resource and to play it back with a call to `SndPlay`.

```
// Play back a snd resource with SndPlay

SndListHandle     sndResource;

sndResource = GetResource('snd ', 1000);
if(sndResource)
{
      OSErr err;

      HLock((Handle) sndResource );
      err = SndPlay( nil, sndResource, FALSE);
      HUnlock((Handle) sndResource );
      ReleaseResource((Handle) sndResource);
}
```

In this example the sound channel parameter of `SndPlay` is nil, which is the signal to `SndPlay` that you are too lazy to build your own sound channel and that it will need to build a temporary one for you. `SndPlay` will only perform this kindness if you ask it to play the sound synchronously. If you attempt to play the sound asynchronously and still ask `SndPlay` to create the sound channel, you'll be indirectly forcing it to ignore you and play the sound synchronously.

From a Disk

One of the cooler things introduced with version 2.0 of the Sound Manager was the ability to play sounds larger than available memory directly from disk. You always could do this before 2.0, it was just that you'd probably prefer an IRS audit to the

task of writing all the interrupt level code necessary to perform such a feat. With 2.0 playing from disk is about as easy as using `SndPlay`.

To play a file from disk you'll first need a sound file to play. Before you scream "Duh!" at me you need to know that you have several choices of sound files to choose from. You can use a resource file with a `snd` resource contained within it. Otherwise if you prefer your sounds in the data fork or if you really miss programming an Amiga you can put your sounds into an AIFF files or its compressed cousin, an AIFF-C file. To convert your sound into an AIFF file you can use any one of the public domain utilities written to convert sound files; my favorite is SoundApp by Norman Franke. Or you can use the program that originally created the sound file. Most can also save the file as AIFF or AIFF-C.

The Sound Manager's function to play back an AIFF file is `SndStartSndFilePlay`. Don't be intimidated by all those arguments in the prototype. Most of them you can set to nil and let the Sound Manager do all the hard work for you. The only two parameters you have to provide are an open file reference and whether you want to play the sound synchronously or asynchronously. You might need to pass in the resource id of the `snd` if you're not playing back an AIFF file. `SndStartFilePlay`, like `SndPlay`, will only let you play the file synchronously if you ask it to build a temporary sound channel for you.

```
OSErr SndStartFilePlay(SndChannelPtr chan,
                       short fRefNum,
                       short resNum,
                       long bufferSize,
                       void *theBuffer,
                       AudioSelectionPtr theSelection,
                       FilePlayCompletionUPP theCompletion,
                       Boolean async);
```

Playing a disk file is then as simple as opening the file, calling `SndStartFilePlay`, waiting for the sound to play, and then cleaning up by closing the input file.

```
// Give an FSSpec play back the AIFF file it points to

short refNum;

if( FSpOpenDF(&fileSpec, fsRdPerm, &refNum) == noErr)
{
      OSErr err;

      err = SndStartFilePlay(nil, refNum, 0, 0, nil,
                                 nil, nil, nil, FALSE);
      FSClose(refNum);
}
```

The processor resources required to play a file directly from disk rule this out as a core technique for high-speed action games. But other types of games without as many processing demands could probably benefit greatly from this nifty and easy-to-use capability.

Asynchronously—Get Used to It

While your game sound effects are playing you still want to have your sprites running around on the screen. This requires that you play all your sounds asynchronously. Asynchronous sounds are played at interrupt level by the Sound Manager while control of the main thread of execution is returned to your game.

Playing a sound asynchronously is easy enough. Create a sound channel. Load in the sound resource. Call `SndPlay`, passing it the sound channel you created, as well as the loaded sound, and setting the function's asynchronous flag to true. If everything goes according to plan `SndPlay` will return immediately to the function that called it and you'll have a sound playing in parallel with your program.

The problem with asynchronous sound (you knew there had to be a problem) is deciding when the sound is done playing. Without this knowledge you would never know when you could start playing your next sound effect. A game that can only play one sound would have to be a very short game.

What you would like is a notification from the Sound Manager when it is done playing your sound. And that is exactly what the Sound Manager provides. When you create your own sound channel you are given the chance to install a "sound done playing" notification function with the sound channel. You would think that your installed callback function would be called as soon `SndPlay` was done playing. You'd think that. But no, that would be too easy.

The callback function associated with your sound channel will only be executed by the Sound Manager when the channel processes a command of the type `callBackCmd`. When the Sound Manager is handed a `callBackCmd` it immediately calls the callback function tied to the sound channel that the `callBackCmd` came from.

This method is a lot more flexible than having a simple completion function called when the channel is done processing sound commands. With this method you can have your callback function executed at any point in the sound processing stream. All you have to do is embed the `callBackCmd` into the channel's playback queue whenever you would like to be called back.

In your case you would like to be called back right after the sound is done playing, simulating a completion routine. This can be done by making sure that you insert `callBackCmd` as the absolute last sound command given to the sound channel. As commands are processed in a FIFO order your callback will be the last command extracted from the sound channel's queue. From this you can see that you must use the `SndDoCommand`, which places commands in the queue, and not `SndDoImmeadiate`, which calls your callback immediately and defeats the whole purpose of the callback.

Let's look at an example that plays a sound resource. The example will keep playing back a sound resource until you click the mouse button or hit the restart button.

```
/* ----------------------------
      Simple driver function to test Asynch sound
---------------------------- */
OSErr PlaySoundForever(short sndID)
```

```
{
      OSErr err = noErr;

      while(!Button() && err == noErr)
      {
            if(gChannel.stillPlaying == FALSE)
                  err = PlaySndAsynch(sndID);
      }

      return err;
}
```

The test function for the asynchronous sound playback takes a resource id of the sound that you would like to hear endlessly. This function watches the `stillPlaying` flag contained within the global `AsynchChannel` structure. When this flag is false the sound channel is done playing the sound and you're free to replay it.

```
typedef struct {
      SndChannelPtr      channelUsed;
      Boolean            stillPlaying;
} AsynchChannel;

AsynchChannel      gChannel = { nil, FALSE };
```

The `AsynchChannel` structure is a convenient way of passing a global flag to the callback without having to worry about A5 worlds.

To play the sound, the test function uses `PlaySndAsynch`, passing it the resource id of the sound it wants played. Before `PlaySndAsynch` starts whipping up a fresh sound channel, the global sound channel reference is checked to see if a channel already exists.

```
/* ------------------------------------------------
      Play a 'snd' asynch with the global channel
------------------------------------------------ */

OSErr PlaySndAsynch(short sndID)
{
      OSErr err = noErr;
```

```
// Create a sound channel for playback
if(gChannel.channel == nil)
      err = SndNewChannel(&gChannel.channel,
                          sampledSynth,
                          0,
                          DonePlaying);
```

If a sound channel does need to be created a call is made to `SndNewChannel`. By passing a pointer to a nil channel pointer, `PlaySndAsynch` is requesting that the Sound Manager create the memory needed for the sound channel and pass back a pointer to it. After the channel pointer you need to specify to `SndNewChannel` what type of synthesizer, or type, of sound you'll be playing through the channel. In this case you'll be playing back sampled sound, so you need to use the sampled synthesizer. Next you pass any channel initialization flags that you want used to configure the sound channel. In this example zero is passed, telling `SndNewChannel` to take its best guess according to the Mac's hardware on how to initialize the channel. The last parameter for creating a sound channel is the address of the function that you want the sound channel to execute whenever it encounters a `callBackCmd`.

The prototype of the callback function must match this declaration. When the callback is executed it is handed a pointer to the sound channel that the callback was associated with and a pointer to a copy of the sound command structure that contains the `callBackCmd`. So along with the command your callback gets handed a pointer to the two parameters that are in every sound command.

```
pascal void CallBack(SndChannelPtr channel,
                     SndCommand * cmd);
```

Using the sound command's parameters is how our callback communicates with the global sound done flag. Our sound callback function, `DonePlaying`, casts the second parameter of the sound command as a pointer to an `AsynchChannel` structure. From this pointer the flag that indicates whether the sound is still playing is set to false. This method, while little more indirect than setting and

restoring the A5 world, is a heck of a lot faster. I always hated messing with A5 anywise.

```
/* ------------------------------------------------
      Sound completion routine
------------------------------------------------ */

pascal void DonePlaying(SndChannelPtr channel,
                                        SndCommand *cmd)
{
     ((AsynchChannel *)cmd->param2)->stillPlaying = FALSE;
}
```

The really important thing you have to remember about sound callback functions is that they operate as interrupt-level. Which makes sense as the callback is called from an interrupt handler. Since the function is interrupt-level code, you can't do anything with the memory manager or resource manager, really anything that would be cool. Just stay away from all Toolbox calls unless you check that they are interrupt-safe, and you should be safe.

```
if(err == noErr && gChannel.stillPlaying == FALSE)
{
      SndListHandle     sndHandle;

      // Load in the resource
      sndHandle = (SndListHandle) GetResource('snd ',
                                              sndID);
      if(sndHandle)
      {
            SndCommand  cmd;

            HLock((Handle)sndHandle);

            // Play the sound
            err = SndPlay (gChannel.channel,
                                 sndHandle, TRUE);

            // Install call back
            if (err == noErr)
            {
                gChannel.stillPlaying = TRUE;
                cmd.cmd = callBackCmd;
```

```
                        cmd.param2 = (long)&gChannel;
                        err = SndDoCommand(gChannel.channel,
                                           &cmd, FALSE);
                    }
                }
            }
            else
                gChannel.channel = nil;

            return err;
}
```

The rest of the function `PlaySndAsynch` loads the sound handle into memory and then makes an asynchronous call to `SndPlay` using the sound channel created earlier. If `SndPlay` returns the all-clear sign, `PlaySndAsynch` sets the global flag `stillPlaying` to true, indicating to the outside world that the sound is currently playing. The callback is initiated by building a sound command that contains a pointer to the `gChannel` structure in its second parameter. This is the pointer that will be passed to your callback function when `SndPlay` is done playing. The command is then inserted into the sound channel's queue with a call to `SndDoCommand`. On inserting the command the function requests that `SndDoCommand` not wait until there is room in the channel's queue. If the sound you passed in happened to have more than the default number of sound commands (around 128) `SndDoCommand` would return an error. You needn't concern yourself, though, as sampled sounds usually use two or three commands at most. Most of the space used by the `snd` resource is dedicated to sample data used by the commands, which don't eat up much of the sound channel's command queue.

Sound Class Kit: "The Audience Is Listening"

For the game class kit to be complete you need an interface for playing sounds. That condition is satisfied with the inclusion of the `CSoundFX` class. This class will manage all the sounds tasks needed by our examples and provide a sound platform for you to build

upon in the future. Sound platform. Get it? Oh never mind, it's late and you have better things to read than bad puns. Like that upcoming bold section heading.

Priorities

About the only new wrinkle that the CSoundFX class brings to sound playback is the idea of sound priorities. Each sound that is played by the class must be assigned one of the priorities defined by the class's interface. This assignment is done at the time the sound is played with the priority usually determined at compile time.

```
typedef enum {
      kLowestPriority = 0
      kMedPriority,
      kHighPriority,
      kExplosionPriority,
      kBonusPriority,
      kAlarmPriority = 255
} SoundPriority;
```

Sound priorities come into play when all of the sound channels are busy playing and you want to play back yet another sound. The CSoundFX class will take your sound and its priority and look for a sound channel that is currently playing a lower-priority sound. If it finds one the class will interrupt the lower-priority sound and start playing back your sound immediately. If your sound ends up having a priority lower than all the sounds currently playing, your play request is dropped into the bit bucket.

When you write your code you have to decide at what priority you want each sound to be played. General background and environment noises should be given the lowest priority. Give average sound effects like missiles shots and laser beams a rating around the mid-range. You want actions that signify accomplishments and disappointments—blowing up an enemy ship or having your ship destroyed—to have a high sound priority. Can't have your player

missing a significant event like that. The highest priority is the `kAlarmPriority`, which should only be used to indicate something really important that the player has to be aware of. Winning a free life would be play back a sound of an alarm priority.

Starting Up

Initializing is done by creating an instance of CSoundFX class. You should do this at the beginning of your program. It would be best if you created only one instance of the class and kept a global reference to it that the rest of the objects have access to. You don't want to be creating a CSoundFX class, playing a sound, and throwing the class away. You could do it that way, but it would be painful.

All of the sound channels used by the class are contained in a wrapper similar to the one used in the asynchronous playing example. It's even called the same thing and used for the same purpose: communicating the state of the channel between the callback routine and the rest of the code.

```
typedef struct {
      SndChannelPtr      channelUsed;
      Boolean            stillPlaying;
      SoundPriority      priority;
      SndListHandle      currSnd;
} AsynchChannel;
```

The first two fields you already know about; the first of the other two tells us the priority of the sound currently playing, with the remaining field keeping a copy of the sound handle that is being played. The handle is kept around so the class can unlock it after it is done playing back.

```
CSoundFX::CSoundFX()
{
      OSErr err = noErr;

      fSoundStopped = FALSE;
```

```
        for(short i = 0; i < kMaxChannels; i++)
        {
            fChannels[i].channelUsed = nil;
            fChannels[i].stillPlaying = FALSE;
            fChannels[i].priority = kNothingPlaying;
            fChannels[i].currSnd = nil;

            // Create all the channels upon creation
            err = SndNewChannel(&fChannels[i].channelUsed,
                                sampledSynth,
                                initMono + initNoInterp,
                                DonePlaying);
            if(err != noErr)
            {
                  PostFatalError(err);
                  break;
            }
        }

        if(err == noErr)
              fSoundStopped = TRUE;
}
```

The constructor for CSoundFX is the only initialization needed to use the class. The constructor creates a number of sound channels for use with the sampled sound synthesizer in the same manner as before, letting `SndNewChannel` create the memory used by the channel.

Each channel is initialized at creation so that it will play back a sound in mono and skip interpolating the samples. Normally a channel will attempt to convert sounds that are sampled at a rate lower than 22 kHz up to 22 kHz during playback by interpolating between the missing samples. With interpolation off, the Sound Manager will instead duplicate existing samples to make up for the missing samples. This does result in audible degradation of sounds sampled at rates lower than 22 kHz, but is much faster than interpolating. You shouldn't be able to notice the difference with noisy sounds like explosions.

The same callback routine is used by all the sound channels. This is one of the benefits of passing at the address of the sound

done flags instead of depending on setting up and global world. To use an A5 world in the manner that *Inside Mac: Sound* suggests, you would need a separate callback function for each channel allocated.

If any problems are encountered during channel creation, the constructor will call `PostFatalError`, the same function used by the play field and sprite classes to report error conditions. This function will call the error handler that you installed at the beginning of the program to handle any sprite or play field problems. So your error handler will now be doing double duty, watching out for any problems with the graphics system and now the sound system.

With all the channels successfully erected, the constructor enables the master sound playback switch by setting `fSoundStopped` to true. It had set this to false at the beginning of the constructor in case of any trouble.

Shutting Down

At the end of your game you'll want to free the sound channels used by the sound class by freeing the class. The class's destructor will handle the rest.

```
CSoundFX::~CSoundFX()
{
      Silence();

      for(short i = 0; i < kNumChan; i++)
      {
            OSErr err;

            err = SndDisposeChannel(
                        fChannels[i].channelUsed, TRUE);

            fChannels[i].channelUsed = nil;
            fChannels[i].stillPlaying = FALSE;
            fChannels[i].priority = kNothingPlaying;
            fChannels[i].currSnd = nil;
      }
}
```

Handling the rest means shutting up any sounds that are currently playing with a call to `Silence`. After any remaining rogue sounds are clammed up, the channels used by the class are then disposed of with a few quick calls to the Sound Manager's `SndDisposeChannel`.

It would have been safe to dispose of the channels without silencing them, as `SndDisposeChannel` is capable of silencing the channel before throwing it on the bit heap. That's exactly what the last parameter tells `SndDisposeChannel` to do. If you pass true as this parameter the channel will be immediately shushed before being disposed of. Pass in false and the channel will wait until it has finished all of the commands in its queue.

Playing Sounds

The main sound playback function of the CSoundFX class is the `PlaySnd` function. All you need to pass to the function is the resource id of the sound you want to play and the priority that you want the sound associated with.

Playing the sound is accomplished by first seeing if the class wants any sound at all to be played with a quick check of the `fSoundStopped` sound-enable flag. If it's true, then the class doesn't want any more sounds being played, so the function bails early.

```
CSoundFX::PlaySnd(short sndID, SoundPriority priority)
{
      SndListHandle     sndH;
      AsynchChannel *   channel = nil;

      if(fSoundStopped)
            return;
```

Next the desired sound is loaded into memory from whatever resource file it lies in. After the sound resource is successfully grabbed, the handle is locked down. Sound handles you don't want wandering around while you're trying to play them back asynchro-

nously. If you want playback to occur reliably you'll want to preload your sounds by either marking them to be preloaded with your favorite resource editor or loading them programmatically at the start of your game or at the beginning of each level.

```
sndH = (SndListHandle) GetResource('snd ', sndID);
if(sndH == nil)
{
      PostFatalError(ResError());
      return;
}

HLock((Handle) sndH);
```

For playback a free channel must be found. First check all the channels to see if any if them are already free. If one is found its address is assigned to the local channel pointer.

```
// Find a channel to play from, first looking for one
// that isn't already busy
channel = nil;
for(short i = 0; i < kNumChan; i++)
{
      if(fChannels[i].stillPlaying == FALSE)
      {
            channel = &fChannels[i];
            break;
      }
}
```

If all the channels are currently busy, the code starts to look for a channel that is playing a sound of lower priority than the one that has been requested to be played.

```
// If all the channels were busy try to find one with
// a lower priority snd
if(channel == nil)
{
      for(short i = 0; i < kNumChan; i++)
```

```
        {
                if(fChannels[i].priority < priority)
                {
                        channel = &fChannels[i];
                        break;
                }
        }
}
```

With a channel found ready for playback, you need to know if it's already busy playing. If so, the channel is silenced by immediately sending a quiet command to the channel. Following that command is the flush command, which wipes all the commands still lying around in the channel's queue.

```
// If a channel was found silence it if necessary and
// then start playback
if(channel)
{
        OSErr       err;

        // Shutdown a channel that is still playing
        if(channel->stillPlaying)
        {
                SndCommand  cmd;

                cmd.cmd = quietCmd;
                cmd.param1 = 0;
                cmd.param2 = 0;
                err = SndDoImmediate(channel->channelUsed,
                                                        &cmd);
                cmd.cmd = flushCmd;
                err = SndDoImmediate(channel->channelUsed,
                                                        &cmd);
        }
```

With the channel successfully shushed and flushed the sound is set to begin playing with a call to `SndPlay`. Other than copying a reference to the sound handle and its priority, this playback code works just like the previous asynchronous example.

Playing back sounds efficiently on the Mac is a matter of keeping the Apple Sound Chip fed and happy. The ASC is happy and at its best when it gets fed regularly and in the right proportions.

Any sound you feed to the ASC that isn't at its natural appetite of 22 kHz it will have to rate-convert. This is time-consuming. The best way to avoid this delay is to always feed the ASC a diet of 22 kHz sounds. The next best substitute is to turn off rate-sampling interpolation when creating the sound channel, as the sample code does.

The ASC has a 1-K appetite. All sound data fed to it is done so in 1-K chunks. When the ASC is done processing half of this chunk it will interrupt the Sound Manager and ask for seconds, thirds, etc. This large appetite means that smaller sounds or sounds that don't end on 1-K boundaries could end up feeding the remainder of a padded buffer before the next sound in the queue can be processed. This padding will be silence and will sound like a pop between sounds.

By feeding the ASC high-nutrition 22-kHz sounds that are larger than 1K you can keep it happily playing 100 percent of the recommended daily allowance of game sounds.

The channel's flags are set before `SndPlay` is called to prevent a potential race condition. This condition could occur if the channel's flag was set after the callback command is set. The race is between the point the callback command is sent to the channel and the amount of time needed to set the callback flags. If the channel's callback function is called before that flag can be set, which could happen for very short sounds, then the channel's callback will execute, marking the channel as no longer busy. When the Sound Manager's interrupt handler finishes up, control will be passed back to the code, which will dutifully mark the channel as busy, even though the callback has already completed. With no chance of a callback, the channel will remain marked as busy and only a sound of a higher priority could ever grab the channel back. Nasty. Try to avoid it.

```
                if(err = noErr)
                {
                        // Show that the sound is playing before
                        // calling SndPlay. This avoids a
                        // potential race condition that would
                        // close off the channel forever

                        channel->stillPlaying = TRUE;
                        channel->priority = priority;
                        channel->currSnd = sndH;

                        err = SndPlay (channel->channelUsed,
                                        sndH,
                                        TRUE);
                        if (err == noErr)
                        {
                                cmd.cmd = callBackCmd;
                                cmd.param2 = (long)channel;
                                err = SndDoCommand (
                                        channel->channelUsed, &cmd,
                                                        FALSE);
                        }
                        else // SndPlay failed for some reason,
                                // mark channel as unused
                        {
                                channel->stillPlaying = FALSE;
                                HUnlock((Handle)channel->currSnd);
                                channel->currSnd = nil;
                                PostFatalError(err);
                        }
                }
                else
                        PostFatalError(err);
        }
}
```

Tidying Up

During your game loop or event loop you'll want to make periodic calls to CSoundFX's `SoundFXTask` function. This function cleans up after any sound channels that are no longer being used.

```
void CSoundFX::SoundFXTask()
{
      for(short i = 0; i < kNumChan; i++)
      {
            if(fChannels[i].stillPlaying == FALSE)
            {
                  // Mark the channel as empty
                  fChannels[i].stillPlaying = FALSE;
                  fChannels[i].priority = kNothingPlaying;
                  if(fChannels[i].currSnd)
                        HUnlock((Handle)
                           fChannels[i].currSnd);
                  fChannels[i].currSnd = nil;
            }
      }
}
```

For our sounds the channels are marked as being completely free. If a sound handle is still hanging around it is unlocked and all references to it are forgotten.

Silence Is Golden

At any point you can silence all playback channels with one call to the class's `Silence` method. From this method all the channels are checked to see if any sounds are currently playing back and if so, they are silenced with a quiet command followed by a flush command.

```
void CSoundFX::Silence()
{
      for(short i = 0; i < kNumChan; i++)
      {
            if(fChannels[i].stillPlaying)
            {
                  SndCommand  cmd;
                  OSErr       err;

                  cmd.cmd = quietCmd;
                  cmd.param1 = 0;
```

```
                cmd.param2 = 0;
                err = SndDoImmediate(
                            fChannels[i].channelUsed,
                            &cmd);
                cmd.cmd = flushCmd;
                err = SndDoImmediate(
                            fChannels[i].channelUsed,
                            &cmd);

                // Mark the channel as empty
                fChannels[i].stillPlaying = FALSE;
                fChannels[i].priority = kNothingPlaying;
                HUnlock((Handle)fChannels[i].currSnd);
            }
        }
}
```

Enabling and Disabling Playback

The only functions of the class left are the ones that enable and disable master playback. By using the functions `Enable` and `Disable` you simply toggle the variable that controls whether any future sounds can be played through the `PlaySnd` method.

12

Digger

In this chapter it all comes together. Here a complete example game is built that uses all the techniques discussed so far plus a few others yet to be covered. The arcade game presented here is a compromise between a full-featured game and a teaching example with enough features to be interesting. Which means that this example is rather large in scope in comparison to the other examples presented earlier. With such a large example only the essential core code of the game will be covered in detail in this chapter. For the rest you can wander around the full source provided on disk.

Game Rules

Digger, the name of this chapter's example, is based on an old arcade game, Dig-Dug. If you've ever played that arcade relic, this game will seem immediately familiar.

The rules for Digger are simple. You control Doug, who loves to dig in his backyard. Trouble is, deep in his backyard are buried alive some ferocious fire-breathing monsters that don't much appreciate Doug's digging around in their territory.

Doug's mission, if he decides to accept it, is to dig a network of tunnels throughout the underground of his backyard and rid himself of these fire-breathing pests. To help with this job Doug is equipped with a freeze ray that his sister built from the spare Nintendo cartridges lying around his garage. By shooting the freeze ray at a monster Doug can start to turn it into a chunk of prehistoric ice. If the ray is applied long enough, the monster will freeze solid and shatter. No more monster. If Doug doesn't freeze a monster solid it will eventually melt the block of ice surrounding it and return to the task of trying to fricassee Doug. Doug's freeze ray has a limited range and takes a while to fully freeze a monster. While Doug is using his freeze ray he has to remain calm and still, which makes him a perfect target for any other monsters still stalking around.

Doug's other advantage is his tunneling speed. Doug is one fast digger. He can dig tunnels slightly faster than the monsters can follow. Doug uses this speed advantage by digging tunnels under the boulders lying around in his yard and moving out of the way in the nick of time. The boulders come crashing down and with luck squish any monsters that were trailing Doug. But if Doug isn't careful the boulders can as easily end up bashing in his noggin. Boulders don't squish people. Gravity does.

The monsters occupying Doug's backyard aren't without their own arsenal. Their first line of offense is some really offensive breath. Every so often these babies can shoot a stream of fire that'll deep-fry Doug in no time. Their other evolutionary advantage over Doug is their ability to change their density and pass right through the earth that makes up Doug's yard. They can only stay in this ghost state for short periods of time, but usually long enough to get Doug a-running.

That's the basic plot line. You control Doug's movements with the four arrow keys. Doug will automatically dig a tunnel wherever he goes. You fire Doug's freeze ray by pressing and holding the space bar. Pause the game with the escape key.

You score a fresh new Doug on the first 10,000 points and every 20,000 more after that. Squishing a monster with rocks is worth more than simply turning them into popsicles. Freezing them from the side is worth more than doing it by sneaking up on them from the top or bottom. But it's more dangerous too, since a monster can only breathe fire to the left or right.

Doug goes on to the next level when all the rocks have been released or all the monsters removed, whichever occurs first. Doug loses a life if a boulder pounds him into the ground or a monster sautés him with fire or even touches him. When Doug loses all his lives the game is over.

Data Structures

The main data structure Digger works with is the `TunnelState`. The play field is divided into a two-dimensional array of cells that hold a `TunnelState`.

```
typedef      enum  {
      Blocked = 0,
      OpenOnLeft = 1,
      OpenOnRight = 1<<1,
      OpenOnTop = 1<<2,
      OpenOnBottom = 1<<3,
      PlayerScent = 1<<4
}TunnelState;
```

As Doug digs his network of tunnels each cell is marked with the proper tunnel state for the path passing through that cell. If Doug makes a long tunnel from left to right all the cells he digs through will be marked as `OpenOnLeft` and `OpenOnRight` except for the end points of the tunnel. Those points will be blocked at one of the ends and will only be marked `OpenOnLeft` or `OpenOnRight` but not both.

A tunnel cell is initially marked as `Blocked`, indicating that no tunneling has occurred in that cell. The state of the tunneling is cumulative upon the cell. A cell that Doug has dug through from top to bottom will first be marked as `OpenOnTop` as Doug digs through the top of the cell. When Doug makes his way through to the bottom of the cell it is then marked as also being `OpenOnBottom`. If at a later time Doug were to dig through the left side of the same cell, it would be indicated by marking the cell `OpenOnLeft`. If Doug continues on to the right side of the cell, the creation of a four-way intersection will be celebrated by tagging the cell as `OpenOnRight`. So with four least significant bits set in this cell, the game's code will know that this cell is open on the top, bottom, left, and right.

The game's code uses this array of `TunnelStates` to determine where the monster can stroll, how far the fire and freeze rays can travel before running into a wall, and whether a rock can start falling and after it starts falling when it runs into the ground.

Tunnel Layout

The game's main data structure for keeping track of the tunnels is maintained by the global `gPathArray`. This is the array that is marked up; Doug moves around within it.

```
TunnelState      gPathArray[kRowCnt][kColumnCnt];
```

This is also the array that will be checked before moving any of the monsters to see if there is a clear path in the direction the monster is trying to move.

One of the limits of using a cell-based data structure like this to indicate play field state is that the player's and monsters' movements must happen on grid boundaries. When you tap the dig right key for Doug he will dig to the right one full cell width as defined by `kGridWidth`. Ditto for monster movement, they'll always advance the distance of one full cell. This would look pretty bad if the sprite representing these characters jumped a whole cell distance every time the sprite moved. Since it would look bad the

game doesn't do that. When a sprite decides to move it moves a few pixels at a time until is has crossed one full cell. To the player the characters are moving smoothly and yet are still constrained to the grid enforced by the play field.

Level Template

Each level of Digger is described by a template resource. This allows each level to be constructed with ResEdit or any other resource editor.

```
typedef struct    {
      short tunnelStartX;
      short tunnelStartY;
      short tunnelEndX;
      short tunnelEndY;
}TunnelRec, *TunnelRecPtr;

typedef     struct      {
      short       tunnelCount;
      TunnelRec   tunnels;
}TunnelLayout, *TunnelsPtr;

typedef Point RockPos;
typedef PointPtr *RockPosPtr;

typedef struct    {
      short rockCount;
      RockPos     rocks;
} RockLayout, *RockLayoutPtr;

typedef Point MonsterPos;
typedef PointPtr *MonsterPosPtr;

typedef struct    {
      short monsterCount;
      Point monsters;
} MonsterLayout, *MonsterLayoutPtr;
```

The level resource (type Levl) starts with a `TunnelLayout` as the header. The first field of the header gives how many tunnels are

preexisting for this level. A tunnel must exist in order to have a monster. You can't have a monster suspended in the middle of bedrock. Following the tunnel count in the resource is a variable length array of `TunnelRec` structures. These structures define the bounds of the tunnel and are expressed in grid coordinates, not pixels. A tunnel record is either vertical or horizontal and cannot be wider or taller than one grid height.

After the tunnels are the rock positions for this level. Like the tunnel structure, the rock structure starts with a rock count and then a variable length array of rock positions. Like the tunnels the rock positions are given in grid coordinates, not pixels.

Following the rocks are the monsters, which are packed into the level resource in the same manner: monster count followed by an array of monster grid positions.

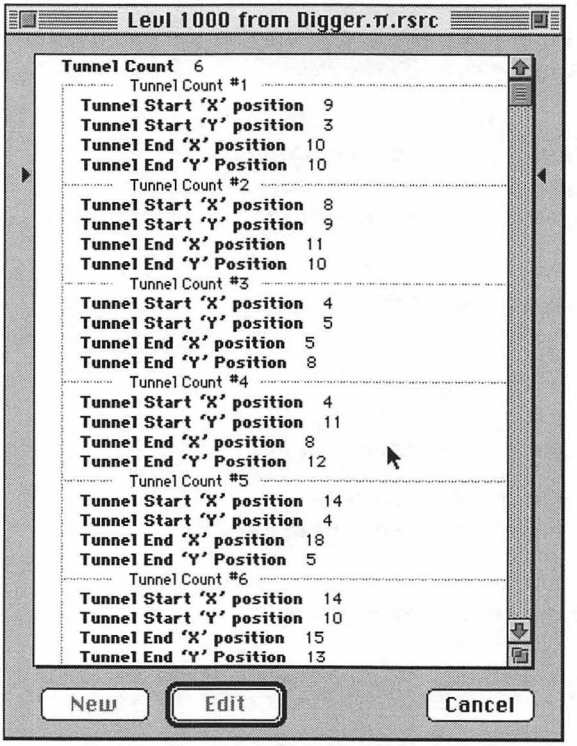


Figure 12-1. Digger levels

With the levels as resources it only takes a few clicks with your favorite resource editor to construct a new level (see Figure 12-1). Much better than encoding all the information in the source code.

Building Levels

At the start of each level of the game the `gPathArray` is initialized by setting all the cells representing dirt to the `Blocked` state. After all the tunnels are cleared, the new level is constructed with a call to `SetupNextLevel`. This function does as its name says, taking the level number and converting it to a resource id of the new level template and constructing that level by parsing the template.

```
void  SetupNextLevel(short currentLevel)
{
      short                   i;
      Handle                  levelHandle;
      TunnelsPtr              tunnels;
      RockLayoutPtr           rocks;
      MonsterLayoutPtr        monstersPtr;
      long                    finalTicks;

      // Get the level layout resource
      levelHandle = GetResource(kLevelResType,
                          currentLevel + kBaseLevelID);
      if(!levelHandle)
            return;
      HLock(levelHandle);
      // Setup ptrs to the elements in the layout
      tunnels = (TunnelsPtr)*levelHandle;

      rocks = (RockLayoutPtr)((*levelHandle) +
            (tunnels->tunnelCount * sizeof(TunnelRec) +
                                        sizeof(short)));

      monstersPtr = (MonsterLayoutPtr)(((char *)rocks) +
                rocks->rockCount * sizeof(MonsterPos) +
                                        sizeof(short)));
```

The first phase of building a level is getting pointers to the tunnel, rock, and monster positions within the level template. This is done by first starting with a pointer to the head of the tunnel section at the start of the resource. The start of the rock section is found by adding the size of the tunnel section to the start of the resource. The tunnel section size is calculated by taking the number of tunnels and multiplying by the size of the tunnel structure and finally adding the size of the tunnel count itself into the equation.

The start of the monster section is found in a similar manner by calculating a pointer that skips over the rock section.

```
// Clear out the tunnels
gDiggerPF->PreDrawOnBackground();

for(i = 0; i < tunnels->tunnelCount; i++)
      DigOutTunnel(&tunnels->tunnels[i]);
gDiggerPF->PostDrawOnBackground();
```

With the proper pointers pointing at the proper things, the code can now start constructing the level. First it loops through all the tunnels in the template and has them dug out with a call to `DigOutTunnel`. Dig out tunnel will mark the cells properly and even blacken in the tunnel that it has dug in the background off-screen from the play field. That is why before the loop was started `PreDrawOnBackground` was called. Wouldn't want the on-screen buffer screwed up. After the tunnels are dug, the proper graphics port is restored with a call to `PostDrawOnBackground`.

```
// Position the monstersPtr in the tunnels
for(i = 0; i < monstersPtr->monsterCount; i++)
{
      if(AddMonsterToLevel(
                  monstersPtr->monsters[i].h,
                  monstersPtr->monsters[i].v,
                        gSpriteGroup ) != noErr)
            break;
}

// Position the rocks
for(i = 0; i < rocks->rockCount; i++)
```

```
{
      if(AddRockToLevel(rocks->rocks[i].h,
                        rocks->rocks[i].v,
                        gSpriteGroup ) != noErr)
            break;
}
```

Then the monsters and rocks are put in their proper positions.

```
gBonusSprite->Hide();
gBonusSprite->SetStartingPosition(kPlayerHorzStart,
                                  kPlayerVertStart);
```

In case the bonus gem sprite was left visible at the end of the previous level, it is hidden and then positioned to its default location.

```
HUnlock(levelHandle);

ReleaseResource(levelHandle);

// Set up the player to the correct position
InitPlayer();

// Force redraw of play field
gDiggerPF->HandlePlayFieldUpdate(
                                MainWindow->portRect);

}
```

With the level resource now done, it is unlocked and set free. The player's sprite is then reinitialized with `InitPlayer`. This function sets back to its proper state for the start of a new level.

The fully constructed level is then copied on-screen through the play field's update mechanism, `HandlePlayFieldUpdate`.

Game Loop

As you can see, this game loop is a little more complicated than the others you've encountered. It has to be. This one handles more situations when a game is paused, when the player loses a life, when the game is over. And moving to the next level.

```
void  RunGameLoop(void)
{
      gGameRunning = TRUE;
      InitPlayer();
      HideMenuBar(gMainWindow);
      HideCursor();
      gDiggerPF->HandlePlayFieldUpdate(
                                    gMainWindow->portRect);
      gSound->Silenece();
      PerformLevelGetReady(2);
```

At the start of the game loop the global, `gGameRunning`, that tells the rest of the game that the game loop is running is set to true. The player's sprite is moved to its starting blocks. And then the screen is prepared for the game to begin by hiding the menu bar and hiding the cursor. The menu bar is hidden so that the Mac looks like a real arcade game and to indicate to the user that the game is currently playing, as well as to indicate that if the player wants to switch to another program he or she must first pause the game. The cursor is hidden to avoid having it flicker or interfere with the game sprites.

In further preparation for the game loop, the screen is updated and currently playing sounds are shut down.

At the start of the game, and the start of every level afterward, `PerformLevelGetReady` is used to warn the player that the level is about to begin. You pass the function the number of seconds you wish it to delay and it waits that long. While it is waiting it puts up a display that prepares the player for the impending melee.

```
while(gGameRunning && gPlayerLives > 0)
{
      if(KeyIsDown(kEscapeKey))
      {
            RemoveGemFromPlayField();
            gSound->PlaySnd(kPauseGame,
                                    kAlarmPriority);
```

```
            gGameRunning = FALSE;
      }
```

The game loop is controlled by two factors, the game running global and the number of the player's lives. The first factor just indicates if the game has been paused or not. If the game is paused you don't want to be going inside the game loop. The resume menu command turns the game back on by simply setting `gGameRunning` to true again.

The second control is the number of lives the player has. When the player's number dwindles down to zero the game is over and there is no need to stay in the game loop.

The first action of the game loop is to check if the user wants the game paused. If so, the code removes any bonus gems from the play field. Why? Because I'm mean. There should be a penalty for pausing. After the bonuses are snatched away, the pause sound is requested to be played and the global pause flag is set to false so that on the next pass of the game loop, pause will finally take effect.

```
if(LevelComplete())
{
      while(RocksAreStillFalling())
      {
            gDiggerPF->MoveSprites();
            gDiggerPF->ShowNextFrame();
      }

      if(!LastLevel())
      {
            ++gCurrentLevel;
            InitPlayField();
            SetupNextLevel(gCurrentLevel);
            PerformLevelGetReady(2);

      }
      else
      {
            InitPlayField();
            CelebrateLastLevelPlayed();
            gCurrentLevel = 0;
            gScore = 0;
```

```
                SetupNextLevel(gCurrentLevel);
                break;
        }
}
```

Next the game loop handles the level checking code. If the current level is complete the game loop waits for all the rocks to finish falling and then checks to see if this was the last level of the game. If not, then the level indicator is bumped up by one and the next level is constructed. Before starting the new level the game is once again paused so that the player can catch his or her breath before starting again.

If the level completed was the last level, the game is over. To handle this case the game loop cleans up the play field and then calls `CelebrateLastLevelPlayed`. Which does just that. After the celebration, which isn't much for this example (but then again there are only three levels so it isn't much a challenge to complete them all), the game is reset back to the starting level and the game loop is exited.

```
        gDiggerPF->CheckForCollisions();
        gDiggerPF->MoveSprites();
        gDiggerPF->ShowNextFrame();
        gSound->SoundFXTask();

        if(gBonusSprite->IsVisible() &&
              (TickCount() - gBonusTimer > (60 * 10)))
              RemoveGemFromPlayField();

        SystemTask();
}
```

With all the level-handling code out of the way, the game loop gets down to the trinity of any game loop: checking for collisions, moving the sprites, and showing the results on-screen. Added to this trinity is the new task of cleaning up after the sound system.

Every pass through the game loop, the program checks to see if the bonus gem is on-screen. The gem should only stay on-screen for the length of `gBonusTimer`. If enough time has passed, the game

needs to deprive the player of one bonus gem. You snooze, you lose.

The Toolbox function `SystemTask` is called as part each cycle through the game loop so that any of those pesky background tasks that need some time, get some.

```
        ShowCursor();
        ShowMenuBar(gMainWindow);

        // If the player runs out of lives,
        // perform a game over celebration
        if(gPlayerLives == 0)
              PerformGameOverCelebration();
}
```

When code breaks out of the game loop the main screen is restored to its proper state by restoring the cursor and the menu bar. Before leaving `RunGameLoop` it checks to see if the player left the game loop by running out of lives, and if so a game-over celebration is performed before returning control back to the main event loop.

Sprite States

You probably noticed that the game loop doesn't have any code that seems to deal with any of the actions of the main character sprites used by the game. You'd be right. All of the actions of the sprites are contained within the sprites themselves.

Each character sprite operates within the game through the use of states. A state is the mode that the sprite is in at a certain point in time. For the player sprite it could potentially be in moving-left state or shooting-a-freeze-ray state or being-squished-by-a-rock state. Everything the player's sprite can do is a state (even standing still is a state), and the sprite can only be in one state at a time. The player can't be moving left while shooting and being squished at the same time. With only one state at a time possible the sprite handles transitioning from one state to the next as the core part of its state handling.

If you're familiar with finite-state machines you'll feel right at home. Well, maybe not right at home unless your home is decorated with state-transition diagrams.

Player

The player sprite will be our first chance to look at a state-driven character. But before diving into the states and their transitions you need to know how the sprites are constructed.

The player sprite has several sprite cels that cover all states that the sprite can be in except firing the freeze ray. The freeze ray is a separate sprite that is managed by the player sprite. Normally this sprite is hidden and only becomes visible when the player hits the freeze ray firing key. When that key is hit the player detects it and positions the freeze ray sprite in its correct orientation with respect to the player's position. Then if the firing key is still down the freeze ray sprite is made visible and starts its journey toward a monster.

Here is a list of all the possible states that the player and freeze ray combo can possibly be in.

```
typedef     enum  {
      movingLeft,
      movingRight,
      movingUp,
      movingDown,
      standingStill,
      shootingLeft,
      shootingRight,
      shootingUp,
      shootingDown,
      retractingLeft,
      retractingRight,
      retractingUp,
      retractingDown,
      playerPushedByRock,
      playerSquishing,
      playerDying,
      killPlayer

}MovementState;
```

The ordering of these states is important, as you'll see. The states are grouped within state groups, which will allow quick testing of where the sprite is currently.

The states of the sprite are maintained through its overridden methods, `Move` and `Collide`, with most of the action happening within `Move`.

```
void CPlayer::Move()
{
      #define      kTunnelWidth      32
      #define      kTunnelHeight     32

      switch(fPlayerState)
      {

            case movingLeft:
            case movingRight:
            case movingUp:
            case movingDown:
                  DigTunnel();
                  break;

            case shootingRight:
            case shootingDown:
            case shootingLeft:
            case shootingUp:
                  ShootRay();
                  break;

            case playerPushedByRock:
                  PushByRock();
                  break;

            case playerSquishing:
                  Squish();
                  break;

            case playerDying:
                  Dying();
                  break;

            case killPlayer:
                  KillOffCurrentPlayer();
            break;
      }
```

The player's movement function first handles the state that it is currently in. Each state set is managed by a separate method within the sprite. After the state switch code is done, the `Move` function starts dealing with key presses that activate the player sprite's movements.

```
if(fPlayerState == standingStill)
{
      if(IsFiringFreezeRay())
      {
            StartFiringRay();
      }
      else if (MovingRightKey())
      {
            fPlayerState = movingRight;
      }
      else if(MovingLeftKey())
      {
            PlayerState = movingLeft;
      }
      else if(MovingUpKey())
      {
            PlayerState = movingUp;
      }
      else if(MovingDownKey())
      {
            PlayerState = movingDown;
      }
}
```

A player can only transition to another state if it is currently standing still. Don't worry, this sounds worse than it looks. At the completion of the movement state in the previous switch code, the state is always reverted to the standing-still state when the player moves to a full grid coordinate.

```
      if(!EqualRect(&fCurrBnds, &fPrevBnds))
      {
            Rect  tunnelRect;

            gDiggerPF->PreDrawOnBackground();
```

```
            tunnelRect.top = fCurrBnds.top + 4;
            tunnelRect.left = fCurrBnds.left + 2;
            tunnelRect.right = tunnelRect.left +
                                        kPlayerWidth;
            tunnelRect.bottom = tunnelRect.top +
                                       kPlayerHeight;

            // Now adjust the top
            tunnelRect.top = Max(tunnelRect.top,
                                       kTopOfGround);

            PaintRoundRect(&tunnelRect,10,10);
            gDiggerPF->PostDrawOnBackground();
        }
}
```

Once all the state and keyboard handling is done, the `Move` function checks to see if any of the previous code bothered to move the sprite. A sprite that moved needs to paint the background black at its previous location to give the illusion that Doug is digging. Before background is painted, a check is made to see if Doug is above ground. Can't dig tunnels above ground. At least not one the player should see.

Moving

The function that handles the sprite movements is `DigTunnel`. This function handles movement of the player. In this extract from the function the code is handling moving to the left. All the other directions are clones of this section with only the names changed.

```
if((fCurrBnds->left - fkeyDownPt.h == 0) ||
      Abs(fCurrBnds->left - fkeyDownPt.h) >= kTunnelHeight)
{
      //Clear the Tunnel section we just left
      ClearTunnel(&fkeyDownPt,OpenOnLeft);

      //Clear the Tunnel wall we are going to move through
      ClearTunnel(TopLeft(fCurrBnds),OpenOnRight);
```

```
        gPlayerState = standingStill;
}
else
        OffsetRect(&fCurrBnds, -kPlayerSpeed, 0);
```

The sprite's current position is checked to see if it has moved one full grid cell to the left from the point the move-left key was pressed. If it has, then the cell array is marked showing that a tunnel has been dug from left to right on the cell that the player currently occupies.

If the sprite hasn't moved a full cell's width yet, it is simply pushed a little more to the left by offsetting the sprite's current bounds.

Freeze Ray

Shooting the freeze ray is handled by the `ShootRay`. This function manages the positioning of the player's water sprite and shooting it. The shooting of the freeze ray is simulated by changing the sprite's current cel index. Each cel is slightly longer than the next until the index `kShootingRightLastFrame` is reached.

This extract from `ShootRay` only shows firing the ray to the right, but all of the other directions are managed in the same manner.

```
if(fWaterSprite->GetCurrentCelIndex() <
            kShootingRightLastFrame && !fFreezingMonster)
{
        Point futurePos;

        fWaterSprite->SetCurrentCel(
                    fWaterSprite>GetCurrentCelIndex() + 1);

        fWaterSprite->MoveTo( fCurrBnds.left + kPlayerWidth/2,
                                            fCurrBnds.top);

        if(!fWaterSprite->IsVisible())
                fWaterSprite->Show();
        futurePos.h = gWaterSprite->fCurrBnds.right;
        futurePos.v = gWaterSprite->fCurrBnds.top;
        if( ! CanMoveToRight(&futurePos, 0))
```

```
        {
                fWaterSprite->SetCurrentCel(
                        fWaterSprite->GetCurrentCelIndex() -1);
                fPlayerState = retractingRight;
        }
}
else if(!fWaterInflating || !fSpaceKeyDown)
        fPlayerState = retractingRight;
```

If the water/freeze ray sprite is not at its rightmost position and a monster is not in the process of being frozen (`fFreezingMonster` shows this), then the freeze ray's cel is moved to the next one. The ray sprite is moved to its correct position next to the sprite. If it isn't already visible it is made so. Next the path of the ray is checked to see if it is a clear shot. A collision with a wall or rock prevents the ray from firing. To show this the sprite state is reversed by setting the player's state to `retractingRight`.

The same retraction state is set if the ray has reached the limit of its range or the ray-firing key was released.

Falling Rock

When a rock collides with the player the collision method will change the state of the players to reflect that the player is now being pushed by a sprite. The `PushByRock` function handles all aspects of the player on its way to being squished.

```
void CPlayer::PushByRock()
{
        TunnelState         section;
        Point               tempPos =   TopLeft(fCurrBnds);

        tempPos.v += fCurrBnds.bottom - fCurrBnds.top;
        section = gPathArray
                [(tempPos.v + kPlayerSpeed) / kGridHeight]
                [tempPos.h / kGridWidth];

        if( fCurrBnds->top >= fMoveExtent.bottom ||
                                        section == Blocked)
```

```
        {
                fPlayerState = playerSquishing;
                SetCurrentCel(kStartPlayerSquishFrame);
        }

        OffsetRect(&fCurrBnds, 0, kRockSpeed);
        SetAutoMoveTime(Max(
                playerSpriteP->moveTimeInterval -1,
                                                kMaxFallRate));
}
```

The whole goal of this function is to detect when the player has finally hit a hard place—the bottom of a screen or a blocked piece of earth. When between a rock and a hard place the sprite's state transitions to the `playerSquishing` state. If the player has not yet fallen to the bottom, it is moved in synchronization with the rate that rocks fall. This gives the illusion of the rock pushing the player down the screen.

Squishing

When the rock is pounding the player into a pancake it is in the squishing state. In this state the function `Squish` is called until the player displays all of its squishing cels; then the player is hidden and the rest of the game is notified of the player's untimely demise with a call to `KillOffCurrentPlayer`.

```
void CPlayer::Squish()
{
        if(fCelIndex < kLastPlayerSquishFrame)
        {
                SetCurrentCel(fCelIndex + 1);
        }
        else
        {
                Hide();
                KillOffCurrentPlayer();
        }
}
```

Dying

Dealing with the player's death is the responsibility of the `Dying` function. Like the `Squish` function (don't you just love these names) this function cycles through the player's death cel sequence before hiding the player's ray gun and the player itself. Finally true death occurs by transitioning to the `killPlayer` state.

```
void CPlayer::Dying()
{
      if(fCelIndex < kLastDieingFrame)
      {
            SetCurrentCel(fCelIndex + 1);
      }
      else
      {
            fWaterSprite->Hide();
            Hide();
            fPlayerState = killPlayer;
      }
}
```

Death

Upon the loss of the one of the player's lives the `Move` function will make a call to the function `KillOffCurrentPlayer`. This function decrements the global count of player's lives by one, and if the player still has any lives left the level is restarted by repositioning the monsters to their original locations. The rocks are not reset. The player's sprite is then reinitialized and the player is given warning that the level is about to start over with the call to `PerformLevelGetReady`.

```
void KillOffCurrentPlayer(void)
{
      gPlayerLives--;

      ResetMonstersToStart();
      if(gPlayerLives)
```

```
        {
                InitPlayer();
                PerformLevelGetReady(2);
        }

        gDiggerPF->HandlePlayFieldUpdate(
                                        &gMainWindow->portRect);
}
```

With the level reset, the screen needs to be updated to reflect this fact to the user before letting the game loop take over again.

Monsters

The monsters are almost complete code clones of the player sprite. They carry around an invisible fire sprite that is positioned when the monster decides it is time to relieve itself. This fire sprite is managed in the exact manner that the player sprite manages its freeze ray sprite. The shootings are retractions done in a similar manner. The only difference is that monster can only spew fire to the left or right while the player can fire the freeze ray in all four directions.

The monsters move in the same manner as the player sprite, sliding along until the sprite reaches a grid point and then deciding where to go next. The only added wrinkle is that the monsters move in the direction of the player's sprite and do not respond to the player's keyboard presses.

Rocks squish monsters as easily as they can squish you. Except that when a monster gets squished you get bonus points; when you get squished you get a headache and a chance to try again.

For the full tour of monsters check out the source file CMonster.cp & .h. Any questions you have will be answered there.

Experiment

That about covers Digger. The best way to discover every nook and cranny of the code is to experiment with it by adding new features or adjusting the existing ones. Some suggestions are listed below.

- Add new monster types
- Create new falling objects, like bowling balls
- Add an instant-replay feature
- Give the player new weapons
- Add new types of bonuses
- Have lava rise up from the bottom if the player takes too long

Or you can skip trying to alter this game and start on your own game.

Game Over

You've made it through the whole book (or you're cheating and you skipped to the end to see who did it) and are just itching to start writing your own games. Great, have at it. The only piece of advice I can offer is always program games you would want to play. That way you're guaranteed to have at least one person who likes it. The only other piece of advaice I have to offer is not to try and create your dream game on your first outing. Too many beginning game programmers start with wanting to create their ultimate fantasy game. "This game will be the coolest. It'll have the action of DOOM but the depth of chess. You'll be able to fly your F-16 from site to site, battling enemy planes all the way. And if your F-16 is shot down you'll have to fight your way out of the prison camp by competing in a martial arts tournament. And it'll have graphics better than Myst, but combined with full-motion video and full 16-bit sound. Oh, and 3D. It'll come with 3D glasses and when you're in

the space battle mode the asteroids will be flying right at you. That'll be nifty." You might want to try and scale down your first attempt, to start with something do-able. Then do another game. After a few more you can start digging out those 3D glasses. But no matter what you decide to program, make sure you have fun doing it. After all, it's only game.

Index

Addison-Wesley warrants the enclosed disk to be free of defects in materials and faulty workmanship under normal use for a period of ninety days after purchase. If a defect is discovered in the disk during this warranty period, a replacement disk can be obtained at no charge by sending the defective disk, postage prepaid, with proof of purchase to:

Addison-Wesley Publishing Company
Editorial Department
Trade Computer Books Division
One Jacob Way
Reading, MA 01867

After the 90-day period, a replacement will be sent upon receipt of the defective disk and a check or money order for $10.00, payable to Addison-Wesley Publishing Company.

Addison-Wesley makes no warranty or representation, either express or implied, with respect to this software, its quality, performance, merchantability, or fitness for a particular purpose. In no event will Addison-Wesley, its distributors, or dealers be liable for direct, indirect, special, incidental, or consequential damages arising out of the use or inability to use the software. The exclusion of implied warranties is not permitted in some states. Therefore, the above exclusion may not apply to you. This warranty provides you with specific legal rights. There may be other rights that you may have that vary from state to state.

Macintosh Programming

Bill Hensler

Sex, Lies, and Video Games

Tired of searching for memory leaks? Bored with tracking down pointers that point to nothing? Here's the book that brings the fun back to writing Macintosh® programs. ***Sex, Lies, and Video Games*** exposes you to the techniques of producing games on the Macintosh—games with captivating animation and objectionable sounds. You don't need any previous game programming experience, or an extensive background in graphics or sound programming, just an ability to write a simple program that involves windows, menus, and other interface elements.

In a light-hearted, unintimidating style, Bill Hensler explores the ins and outs of arcade-style programming in C and C++, introducing basic game theory, animation, sound, and interaction techniques. Along the way you'll learn the arcane lore of game programming, including:

- sprite blitting
- off-screen and on-screen animation
- sprite cels and directing sprite movement
- ins and outs of producing sounds within your games.

You'll also learn how to put it all together in the full-blown arcade game that's included on the enclosed disk. Now all Macintosh programmers can satisfy their secret desire to become game programming mavens.

During the day, **Bill Hensler** programs video editing software. At night he has been known to write books, play way too many computer games, and sneak up on perfect strangers and ask them to "Pick a card. Any card."

Cover design by Rachel Rutherford

Addison-Wesley Publishing Company

Find A-W Developers Press on the World-Wide Web at:
http://www.aw.com/devpress/

ISBN 0-201-40757-4

$34.95 US
$48.00 CANADA