


THE AMIGA™

Images,
Sounds, and
Animation
on the
Commodore®
Amiga™

MICROSOFT
P R E S S

Michael Boom

ENDORSED BY  AMIGA.

THE AMIGA™

THE AMIGA™

Images,
Sounds, and
Animation
on the
Commodore®
Amiga™



Michael Boom

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
16011 N.E. 36th Way, Box 97017 Redmond, Washington 98073-9717

Copyright © 1986 by Michael Boom
All rights reserved. No part of the contents of this book
may be reproduced or transmitted in any form or by any means
without the written permission of the publisher.

Library of Congress Cataloging in Publication Data
Boom, Michael.

The Amiga: Images, Sounds, and Animation on the Commodore Amiga.
Includes index.

1. Amiga (Computer)—Programming. 2. Computer graphics. 3. Computer sound
processing. I. Title.

QA76.8.A177B66 1986 006.6'765 86-16388
ISBN 0-914845-62-4

Printed and bound in the United States of America.

2 3 4 5 6 7 8 9 MLML 8 9 0 9 8 7 6

Distributed to the book trade in the United States by Harper & Row.
Distributed to the book trade in Canada by General Publishing Company, Ltd.

Distributed to the book trade outside the United States and Canada by Penguin Books Ltd.
Penguin Books Ltd., Harmondsworth, Middlesex, England
Penguin Books Australia Ltd., Ringwood, Victoria, Australia
Penguin Books N. Z. Ltd., 182-190 Wairau Road, Auckland 10, New Zealand

British Cataloging in Publication Data available

Apple™ is a trademark of Apple Computer, Incorporated, and Macintosh™ is a trademark licensed to
Apple Computer, Incorporated.

LIVE!™ is a trademark of A-Squared Systems.

Atari® is a registered trademark of Atari, Incorporated.

Casio® is a registered trademark of Casio, Incorporated.

Amiga™, Amiga Kickstart™, Amiga 1300 Genlock System™, Amiga Workbench™, AmigaDOS™,
Graphicraft™, and Textcraft™ are trademarks of Commodore-Amiga, Incorporated.

Amiga BASIC™ is a trademark of Commodore-Amiga Incorporated and Microsoft Corporation.

Commodore® is a registered trademark of Commodore Electronics Limited.

Centronics® is a registered trademark of Data Computer Corporation.

DeluxeMusic®, DeluxePaint®, DeluxeVideo®, Electronic Arts®, and Instant Music® are registered
trademarks of Electronic Arts.

Epson® is a registered trademark and JX-80™ is a trademark of Epson America, Incorporated.

IBM® is a registered trademark of International Business Machines Corporation.

Microsoft® is a registered trademark of Microsoft Corporation.

Mimetics™ and SoundScape™ are trademarks of Mimetics Corporation.

Motorola® is a registered trademark of Motorola, Incorporated.

Digi-View™ and NewTek™ are trademarks of NewTek.

Okimate 20™ is a trademark of Okidata, a division of Oki America, Incorporated.

Pioneer® is a registered trademark and LaserDisc™ is a trademark of Pioneer Electronic Corporation.

Sony® is a registered trademark of Sony Corporation of America.

Ministudio® and Portastudio® are registered trademarks of Teac Corporation.

The Micro Forge® is a registered trademark of The Micro Forge.

Aegis Animator™ and Aegis Images™ are trademarks of The Next Frontier Corporation.

Space Archive™ is a trademark of Video Vision Associates Limited.

Bugs Bunny® is a registered trademark of Warner Brothers, a division of Warner Communications
Incorporated.

Diablo® and Xerox® are registered trademarks of Xerox Corporation.

DEDICATION

This book is dedicated to all the characters at Amiga, who designed a great computer and kept me very entertained for a year and a half.

TABLE OF CONTENTS

Foreword		<i>ix</i>
Acknowledgments		<i>xi</i>
Introduction		<i>xiii</i>
SECTION 1	THE MACHINE	
Chapter One	A Close Look at the Amiga	3
SECTION 2	IMAGES	
Chapter Two	A Video Graphics Primer	27
Chapter Three	Amiga Graphics Tools	53
Chapter Four	Amiga BASIC Graphics: Screens, Windows, and Palettes	85
Chapter Five	Amiga BASIC Graphics: Creating Images	113
Chapter Six	Amiga BASIC Graphics: Odds and Ends	143
SECTION 3	SOUNDS	
Chapter Seven	An Electronic Music Primer	173
Chapter Eight	Amiga Music Tools	195
Chapter Nine	Amiga BASIC Sound: Music and Speech	221
SECTION 4	ANIMATION	
Chapter Ten	A Computer Animation Primer	251
Chapter Eleven	Amiga Animation Tools	269
Chapter Twelve	Amiga BASIC Animation: Creating Moving Objects	295
Chapter Thirteen	Amiga BASIC Animation: Controlling Motion	327
Afterword	The Future: Amiga's Creative Possibilities	351
Appendix A	Amiga BASIC Statement Formats	354
Appendix B	Companies Mentioned in This Book	360
Index		363

FOREWORD TO *THE AMIGA*

Electronic media diversify our world by giving us almost instantaneous access to a multiplicity of experiences. In the course of a few hours, we can listen to the Beatles, talk to a friend on another continent, watch the Super Bowl, and be in Vienna with Salieri and Mozart.

In nearly every case, the popularity and cultural impact of an electronic medium have paralleled its degree of realism of sights and sounds: The best electronic media put real life in a box. Personal computers have rarely competed with other electronic media in the realm of realistic sight and sound reproduction, but the Amiga's video and sound are new dimensions in the typically number-crunching world of computing. Marshall McLuhan said, "The medium is the message." And the Amiga's message is audiovisual.

The Amiga is the first personal computer that can begin to approach the video quality of television and the sound quality of hi-fi. But unlike television and hi-fi, the Amiga is an interactive medium. With television, you watch another world through the video window; with the Amiga you can step through that window and be in the other world. With hi-fi, you listen to someone else's music; with the Amiga you can change that music or create your own. We're talking about more than just watching and listening; the Amiga is a medium of doing.

The Medium of Doing has arrived just in time. In the past twenty-five years, while generations have grown up in front of the "boob tube," Dr. Marian Diamond has been doing brain research at the University of California. The results have been tabulated, and Dr. Diamond has proven that the single best way to increase your intelligence is through interaction with your environment. In short, "use it or lose it." A Chinese proverb puts it another way: "I hear and I forget, I see and I remember, I do and I understand."

Learning requires motivation. McLuhan must have had visions of the Amiga when he said, "Those who draw a distinction between education and entertainment don't know the first thing about either." The beauty of the Amiga is that you can learn by doing, while being entertained by realistic and dazzling sound and video. Our natural urges to create, to experiment, and to explore are given tremendous new power by the Amiga.

The video games of a few years ago displayed images, made primitive sounds, and were interactive, but they proved to be a passing fad because the medium lacked audiovisual realism and the subject matter was not of enduring human interest. We now know that great software needs to be simple to learn, audiovisually "hot" to grab and keep our attention, and deep and relevant enough that our minds are engaged. With its ease of operation and

tremendous powers, the Amiga is an ideal tool for the software artist who wants to establish new standards in all of these areas. Software that brings the best out of the Amiga will boggle minds and make people squeal with delight.

Can we even imagine the future possibilities for the Amiga? Western Union could have owned the patent on the telephone, but turned it down; the company couldn't see the potential. Television was first thought to be suitable only for civil-defense communications. Only time and creative minds will determine the true potential of the Amiga.

Much has been made recently of "desktop publishing," which, by improving the quality and control available to the individual, may revolutionize the process of communicating on paper. Without discounting the importance of paper, video and sound loom as the media of choice when you have to get a message across. It stands to reason, then, that a "desktop video" market may develop around the Amiga that eventually surpasses desktop publishing.

As the first audiovisual personal computer, the Amiga may be the innovative product that stimulates an intellectual revolution, akin to Gutenberg's printing press. *The Amiga: Images, Sounds, and Animation*, thoughtfully written by Michael Boom, can help you have your own part in that revolution. Viva Amiga!

A handwritten signature in black ink, appearing to read "Trip Hawkins". The signature is fluid and cursive, with a prominent initial "T" and a long, sweeping underline.

Trip Hawkins
President, Electronic Arts

ACKNOWLEDGMENTS

A book that touches on as many different subjects as this one does could not be written without research help from many people. I'd like to thank (in no particular order) the many folks at Amiga who taught me so much about the machine they made and love: Sam Dicker, Carl Sassenrath, Bill Kolb, Dave Needle, Howard Stolz, Rick Geiger, Mitchell Gass, Stan Shepard, Rick Rice, Caryn Havis, Neil Katin, Jack Haeger, R. J. Mical, Barry Whitebook, Terry Ishida, Bob Pariseau, and many others who answered questions in the middle of incredibly busy production schedules. At Electronic Arts, I'd like to thank David Grady for information about Deluxe Paint, and Stewart Bonn for feeding me beta copies of Deluxe Video, showing me Instant Music, and letting me badger him for some of the wonderful Electronic Arts computer games.

Many companies lent me equipment to use while I wrote this book, a great help for a writer on a limited budget. I'd like to thank Jerry Kovarsky at Casio, Bill Mohrhoff at Teac, and especially Jeff Stenehjem at Xerox for time and information above and beyond the call of duty.

Of course, research is the fun part of a book. The truly arduous task is writing it and getting it ready for publication. I'd like to thank the talented staff of Microsoft Press for helping me with all aspects of producing this book, especially Dave Rygmyr for shepherding the manuscript from the first chapter to the final blues with unfailing cheer, skill, and true interest in the contents of the book, and Allan McDaniel and Tom Corbett from over in Microsoft software for answering last-minute Amiga BASIC questions.

Finally, I'd like to thank Lynn Morton for support and understanding through obsession, exasperation, and exhaustion. Thanks, honey, it's done now. We can relax.

INTRODUCTION

If you ask people what their favorite art medium is, chances are excellent that they won't reply "personal computers." Most people think personal computers are fine for managing a database, running a spreadsheet, and processing a few words here and there, but they usually don't think of them as an expressive medium. If they do, they rank personal computers somewhere below crayons and kazooes for creating quality images and sound.

Amiga users know this stereotype is false. With internal hardware and system software designed specifically for graphics and music, the Amiga juggles colors and sound as easily as it calculates spreadsheets and shuffles words. By using the powerful creative software available for the Amiga, users can draw pictures with exquisite color and detail, and produce music and sound effects on a par with professional synthesizers. Amiga users can combine their artistic and musical masterpieces to create cartoon-quality animation, an activity that was previously unavailable to all but professional and student animators with access to expensive equipment.

Like any other medium, using the Amiga creatively takes knowledge and practice. This book is written to give you both, teaching you the concepts you need to understand what you're doing, and giving you examples so you can experiment with what you learn. If you're a novice Amiga user, you'll find the fundamental concepts of computer graphics and sound explained in primers throughout the book. With these concepts in hand, you can easily go through the following chapters that show you how to use application programs like Deluxe Paint, Deluxe Music, and Deluxe Video. Step-by-step examples give you practical experience using these applications.

Experienced Amiga users will also find much of interest in this book. The chapters on application programs explain advanced features, and show you some tricks that you can use to get unique results. They also review other software on the market, and explain how additional hardware can turn your Amiga into an even more advanced creative tool. If you're a BASIC programmer, you'll find chapters explaining the intricate details of Amiga BASIC graphics, sound, and animation commands, and showing you in example programs how to combine them for useful results.

HOW THIS BOOK IS ORGANIZED

This book is divided into four sections to make it easy for you to find a specific area of interest. Section 1 introduces you to the Amiga, taking you on a detailed tour of its hardware and system software to see what makes it such a powerful computer. Section 2 covers graphics, showing you how to create still images on the monitor screen. In Section 3, you'll find information about sound—how to create music, speech, and sound effects using the Amiga's internal sound synthesis. Section 4 teaches you about animation, showing you how to give motion to figures on the screen.

Each of the last three sections is organized the same way: The first chapter of each section is a primer that introduces you to important concepts you'll work with in that section. For example, you learn about how computer monitors create images in Section 2, how the ear hears sound in Section 3, and how the illusion of motion is created by changing still pictures in Section 4. The second chapter of each section shows you how to use a specific application program—Deluxe Paint in Section 2, Deluxe Music in Section 3, and Deluxe Video in Section 4—and discusses additional software and hardware you can use with your Amiga. The remaining chapters in each section deal with Amiga BASIC statements and functions.

HOW TO USE THIS BOOK

The first section of this book was written to give you a full understanding of the parts of the Amiga, so you won't be confused by descriptions in the following three sections. If you know the Amiga well already, you might want to skip Section 1 and jump directly to whichever of the last three sections interests you most.

Within each special-interest section, you should read the first chapter—it introduces you to concepts you'll need to know to understand the other chapters in the section. Once you've read the first chapter, you can read the second chapter and skip the BASIC chapters that follow it if you're interested in application programs and not BASIC programming, or you can skip the second chapter and read just the BASIC chapters if you're interested only in BASIC programming. If you haven't used Amiga BASIC before, you should be sure to read Chapter Four (in Section 2); it tells you how to enter and run programs.

You might want to try reading some of the BASIC chapters even if you aren't a programmer. Much of the information there is interesting to non-programmers, and if you read the beginning of Chapter Four, you'll learn enough to enter and run the BASIC program examples without having to know how they work. In the interest of easy typing and less chance for error, the program examples (with a few exceptions) are short and simple. Just remember to press the RETURN key at the end of each line as you type a program to enter the line in the Amiga's memory.

There are two helpful appendices in the book. Appendix A is a list of all the BASIC statements and functions used in this book, with the format they use and a short explanation. As you write your own BASIC programs, you can use this appendix as a convenient reference if you forget the exact format of a statement, or if you need to find the right statement to perform a BASIC task. Appendix B is a list of all the software and hardware companies mentioned in the book, with addresses you can write to for more information about their products. Following these appendices is a full index to help you quickly find any topic discussed in the book.

WHAT YOU NEED TO USE THE EXAMPLES IN THIS BOOK

To try the examples in this book, you need four different software packages: Deluxe Paint for Chapter Three, Deluxe Music for Chapter Eight, Deluxe Video for Chapter Eleven, and Amiga BASIC for all the BASIC chapters. Amiga BASIC comes with every Amiga; you can buy the other three programs at any Amiga dealer.

You can run the examples with a minimum of equipment. Although some of the examples that show you how to record your creations require a camera, a printer, a VCR, or a cassette recorder, most require no more equipment than your Amiga, monitor, and mouse. Since it's quite easy and inexpensive to add memory to the Amiga, all the examples in this book assume that you're using an Amiga with 512 kilobytes of RAM—they may not work on a 256K Amiga.

One important note: If you're a rank beginner at using the Amiga and don't know how to use the mouse, menus, keyboard, and its other parts, you should read *Introduction to Amiga*, a manual that comes with the Amiga, before you read this book. It describes all the basics you need to know to use the Amiga. Once you know them, you should have no trouble.

Time spent in using the Amiga's sound and graphics creatively can be some of the most enjoyable time you'll ever pass in front of a computer. The things you learn in this book should help you turn your ideas into visible and audible results with a minimum of trouble. Your friends may laugh when you sit down at the keyboard, but when you start to roll the mouse. . .

SECTION 1

The Machine

In this section, you take a close look at the Amiga to see what it is and what it does. You'll learn about the equipment the Amiga uses to run, and to produce graphics and sound. You'll also learn about the different levels of software included with every Amiga. The information and concepts presented in this section will take some of the mystery out of the Amiga's performance, and give you the background to tackle other sections of this book with ease.

The background of the entire page is a complex halftone pattern. It consists of a grid of small dots, with some dots missing or faded to create a series of overlapping, irregular shapes that resemble stylized flowers or leaves. The pattern is symmetrical and repeats across the page. In the center of the page, there is a square hole. Inside this hole, there is a smaller, solid white square. The text 'CHAPTER ONE A CLOSE LOOK AT THE AMIGA' is printed in a bold, sans-serif font within a white rectangular box in the upper right quadrant of the page.

**CHAPTER ONE
A CLOSE LOOK AT
THE AMIGA**

When you first encounter an Amiga, you see only half of what makes it work. The half you can see and touch—the keyboard, the console, the monitor, and the disk drive—is the hardware. The invisible half of the Amiga is its software.

Software is a set of instructions, stored in the computer or on a floppy disk, that determines how various components and functions interact—how the keyboard sends messages to the computer, for example, and how the monitor screen displays pictures. The software makes the computer perform tasks that are helpful to you. Without it, a computer is an expensive and inactive desk ornament.

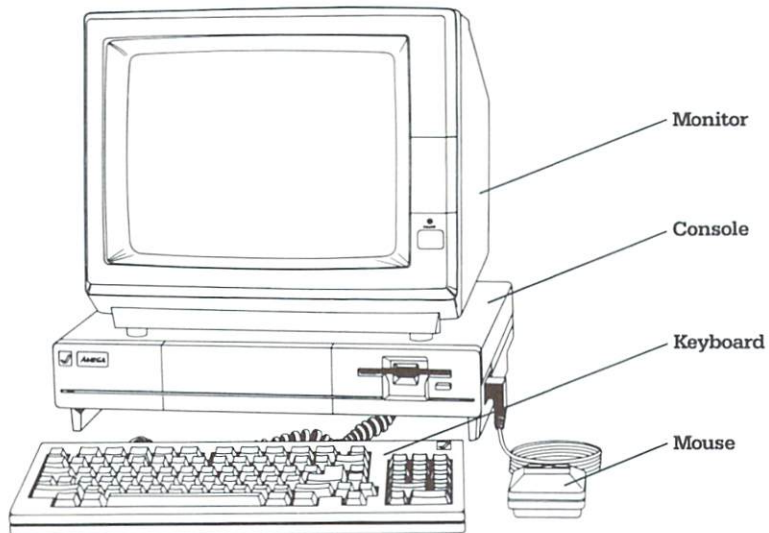
A complete Amiga computer system is a working partnership of both hardware and software. To understand the entire system, you must learn about both halves of the computer.

THE HARDWARE

In Figure 1-1, you see the four main components of a basic Amiga computer system: the console, the monitor, the keyboard, and the mouse. You can add other useful components (called peripherals) to the Amiga, such as printers, modems, or additional disk drives, until your system expands beyond your desk and starts to take over the room. However, for the minimalist or the frugal, these four essential components are the bare minimum you need to make the system work.

Figure 1-1.

The four basic hardware components of the Amiga computer system.



THE MONITOR

Chances are you pay more attention to the monitor than to any other part of the Amiga system. The monitor is the Amiga's channel of communication with you; it gives you visual information through its screen, and audio information through its speaker.

The Amiga can work with three different kinds of monitors: RGB monitors, composite video monitors, and standard television sets. RGB (short for Red, Green, Blue) monitors show the sharpest picture with the most vivid colors, and can display 80 columns of readable text. You should use an RGB monitor with the Amiga, even if you have to save pennies for years, because it's the only monitor that does full justice to the Amiga's graphic abilities.

A composite video monitor is midway in quality between an RGB monitor and a TV set. If you have a component television system, chances are the monitor that displays the picture is a composite video monitor. While it doesn't display as clear a picture as the RGB monitor, a composite video monitor attached to the Amiga still produces a good-looking picture, thanks to the Amiga's superior composite video signal.

The standard home television set can range in quality from pretty good to horrible. The advantage to a TV set, of course, is that you can use your regular TV without having to buy a special computer monitor. If you do use a color television set for a monitor, the Amiga gives it an excellent TV signal for color display. On a good-quality TV set, the Amiga can display 60 columns of readable text on the screen.

The Amiga has three different connectors (called ports) for connecting monitors, one port for each kind of monitor. Since the three ports all send out the same display information (although in different forms), you can connect all three different kinds of monitors at the same time and show the Amiga's video display on three screens at once.

THE KEYBOARD

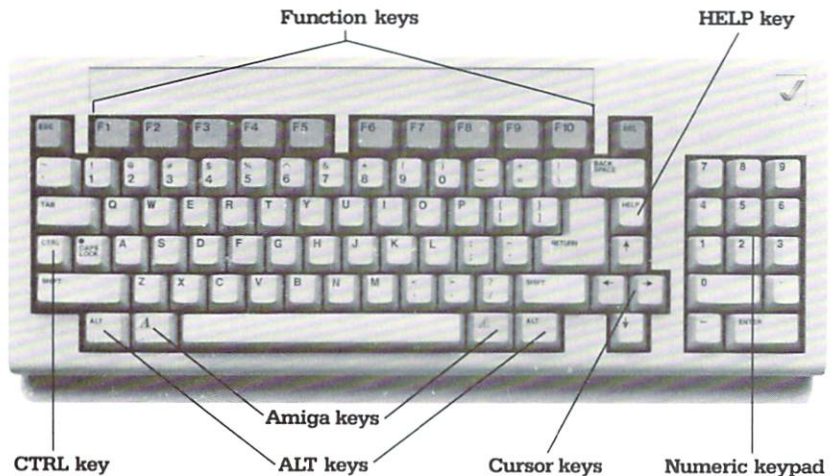
You use the keyboard to send information to the Amiga, feeding it words, numbers, or requests for special functions as needed. The keys are laid out like a standard typewriter, so if you're a touch typist, you'll feel at home.

You'll also notice some keys not found on a typewriter. On the left and right sides of the Spacebar you'll find the two Amiga keys, each marked with a red "A." By holding down an Amiga key and pressing another key, you can perform special program functions, such as choosing a menu item. Next to the Amiga keys are the ALT keys, and above the left SHIFT key you'll find the CTRL key. The ALT and CTRL keys are special keys that, like the Amiga keys, are used in combination with other keys to execute various program functions.

To the right of the main keyboard are four cursor keys, marked with arrows, that let you move the cursor around the monitor screen in applications such as word-processing programs. Above the cursor keys is the HELP key, which you can use in some applications to get helpful information about using that program. Along the top of the keyboard are ten function keys (F1-F10) that can perform special program functions with a single keystroke, and on the far right of the keyboard is a numeric keypad that's laid out like a calculator to speed number entry. You can see these special keys in Figure 1-2.

Figure 1-2.

The special keys of the Amiga keyboard.



The keyboard is connected to the Amiga with a coiled extension cord so you can move it to the location most comfortable for you—a considerable convenience if you like to type lying down or scrunched over in weird positions. The keyboard has a set of small legs in the back that you can fold up to slide the keyboard under the console so that it will be out of the way when you're not using the Amiga.

THE MOUSE

The mouse does away with a lot of typing and is the key to the simplicity of using the Amiga. As you roll it on a flat surface, the mouse moves an on-screen pointer. You can use the mouse to move the cursor quickly in word-processing programs, for example, or to select menu items or objects on the screen that start activities like printing or displaying different sections of text. The mouse also makes an excellent drawing instrument: You can use it with graphics programs to draw images on the screen.

THE CONSOLE

The console is the real workhorse of the system. It contains all the microchips and other electronic components that actually do the work of computing and storing information. The console also contains the internal disk drive, which you use to access and store data and programs on floppy disks. From the outside, the console looks simple and innocent. You have to look inside to see what a hotbed of activity the console really is.

The interior of the console is not the rat's nest of tangled wires and glowing lights that you might expect if you've watched a lot of low-budget science fiction movies. Instead, it's an orderly array of microchips and other small electronic components, all laid out on two green boards collectively called the motherboard. You can see the motherboard and other components that are housed in the console in Figure 1-3.

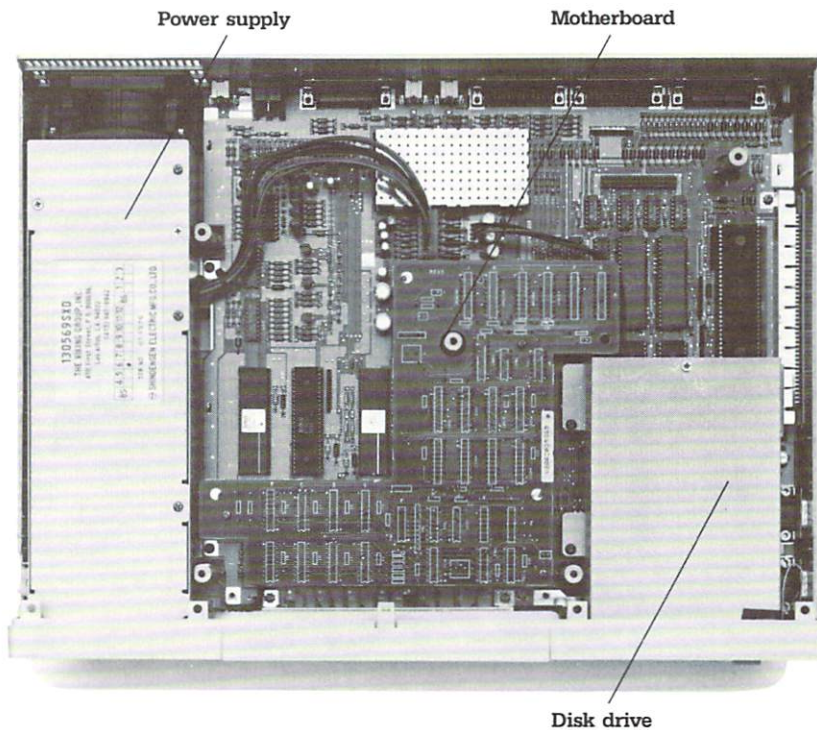


Figure 1-3.

Inside the Amiga console: the motherboard, internal disk drive, and power supply.

The power supply

The power supply simply converts the electricity from your wall socket into the small voltages required to run the Amiga. A fan in the power supply keeps the inside of the console cool; that's necessary because when the computer is on, shuffling electrons

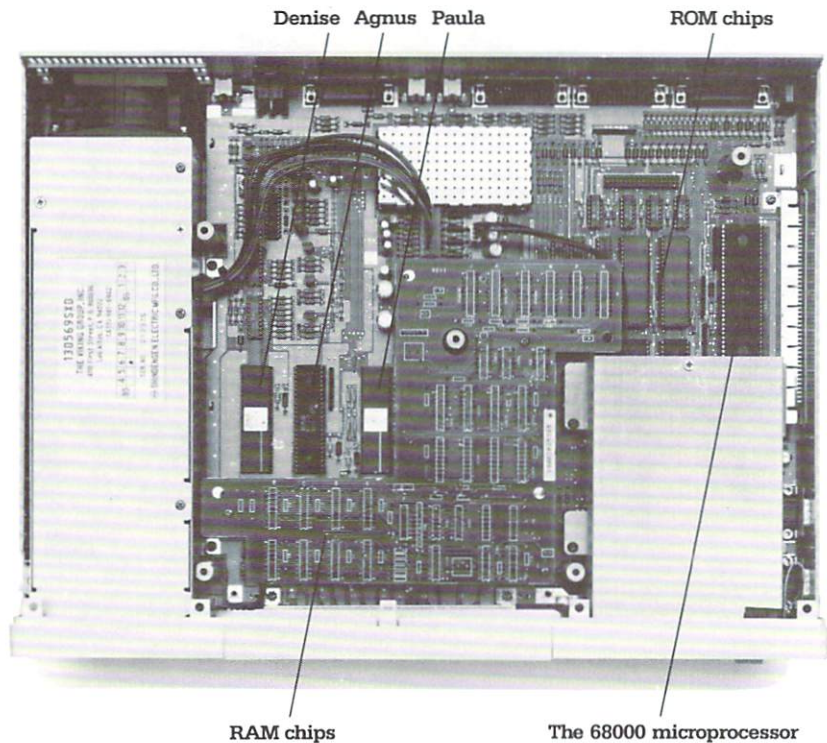
through its circuits, it generates heat. Too much heat can keep the chips from working correctly, and could possibly damage them. To help keep the computer cool, there are vent slots on the bottom and back of the console. You should take care not to block these slots when you use the Amiga.

The motherboard

Most of the real work of the Amiga is done by just four chips on the motherboard: the microprocessor and the three custom chips (see Figure 1-4). The microprocessor is the master chip that does most of the "brain work" of the Amiga such as adding, dividing, and sending commands and information to other chips. The custom chips, which are unique to the Amiga and contribute significantly to its overall computing power, back up the microprocessor by taking care of details such as creating video displays and generating sounds. This frees up the microprocessor chip and allows it to run more efficiently.

Figure 1-4.

The chips on the motherboard: The microprocessor chip, custom chips, and RAM and ROM chips are labeled.



The three custom chips go by the names of Agnus, Denise, and Paula. The microprocessor chip goes by the more imposing name of the Motorola 68000.

The Motorola 68000 microprocessor

The power of a microprocessor is measured by the amount of data it can handle at a time and the speed at which it operates. As microprocessors go, the 16/32-bit Motorola 68000 is fast. 16/32 indicates the 68000's data-handling capabilities: The 68000 processes data in chunks of 16 bits and 32 bits. Inside the microprocessor, the 68000 works on 32 bits of data at a time, then sends data out to the rest of the computer in 16-bit chunks. This makes the 68000 much faster than a 16-bit microprocessor, such as that used by the IBM PC, and makes the Amiga a very quick machine.

Other computers, such as the Apple Macintosh and the Atari ST computers, also use the 68000 microprocessor. What sets the Amiga apart from these computers is its set of custom chips that give it even greater speed and enhanced abilities.

The custom chips

Each of the custom chips has its own set of duties. Denise is the display chip; it controls the way images are created on the monitor screen. Agnus is the animation chip; it draws and moves figures on the monitor screen. Paula is the peripherals/sound chip; it creates the Amiga's sounds, and also sends data back and forth between the motherboard and the peripherals, such as the disk drive, mouse, and keyboard.

The custom chips are perfect subordinates to the microprocessor. They can perform their duties without bothering the 68000, leaving it free to concentrate on weightier matters like numerical calculations. This is one of the truly revolutionary features of the Amiga hardware. Many other computers have no custom chips, so the microprocessor must do all the display, sound, and peripheral work, which slows it down considerably.

Another feature of the Amiga is the way in which its microprocessor interacts with its custom chips. Most computers that have custom chips usually require the microprocessor to constantly supervise the custom chips, which again slows down the microprocessor, although not nearly as much as a computer without custom chips. In the Amiga, however, the microprocessor is interrupt driven, which means that the system was designed so that the custom chips send a signal to the microprocessor when they need instructions, rather than the microprocessor having to constantly check each chip (a process called polling).

The Amiga's interrupt-driven microprocessor contributes significantly to the speed of the system. Together with the custom chip partnership, it gives the Amiga the necessary speed to animate figures smoothly on the screen, transfer data quickly from a disk to the internal computer, or to perform business calculations without making you wait a long time. The chip partnership also means that many of these tasks can be performed simultaneously.

The memory chips

Memory chips are another important set of chips on the motherboard. There are two types of memory chips: RAM and ROM. RAM is short for Random Access Memory, ROM is short for Read Only Memory. Each type of memory stores data, but in a completely different manner.

RAM, also called read/write memory, is erasable, so the microprocessor and custom chips can store data there, retrieve it later, and erase the data so new data can come in. You should note that nothing can be kept in RAM permanently, because RAM has the unfortunate characteristic of losing all of its stored data when the power is turned off.

ROM isn't erasable; the data stored there (sometimes called firmware) is always available to the Amiga. The microprocessor and custom chips can read the data in ROM, but they can't erase it or store new data there. The Amiga's ROM chips store important system software (put into them at the factory) that the computer uses to run itself (a process called booting) when you first turn it on.

Other chips and electronic components on the motherboard take care of details needed to send data and audio-video information out of the console. Although they're all necessary, the microprocessor, custom chips, and memory chips are the most important chips on the board.

The internal disk drive

The disk drive built into the console is called the internal disk drive, to distinguish it from drives that can be attached to the console by cable (conveniently called external disk drives). Disk drives are necessary to read data stored on floppy disks, and to store the data that would otherwise be erased from RAM when you turn off the computer.

The Amiga's internal drive uses 3½-inch floppy disks, a convenient size to put in a shirt pocket and carry around. The internal drive is a double-sided drive, which means that it records data on both sides of the disk. This gives each disk twice the data capacity that it would have on a single-sided drive. The Amiga drive offers 876 kilobytes of storage on each disk, a tremendous capacity for a disk that small.

The Amiga uses a special method called direct memory access (DMA) to store and retrieve disk data. Simply put, DMA means that data can go directly from the disk drive to the Amiga's memory or from the Amiga's memory to the disk drive without passing

through the microprocessor. This means that data transfer between the drive and memory is very fast. It also means that the disk drive can transfer data to and from RAM without stopping the other workings of the Amiga. This is very handy when the Amiga is busy animating a figure on the screen or creating music on the speaker. It doesn't have to stop while it gets new information from the disk drive for the next animation sequence or a new song.

PORTS AND CONNECTORS

Arranged around the outside of the console are sockets for connecting peripherals. These are called ports and connectors, and act as portals for passing information between the chips inside the Amiga and equipment in the outside world. In Figure 1-5, you can see the ports and connectors on the front and right side of the Amiga console. In Figure 1-6 (on the next page), you can see the ports and connectors on the back of the console.

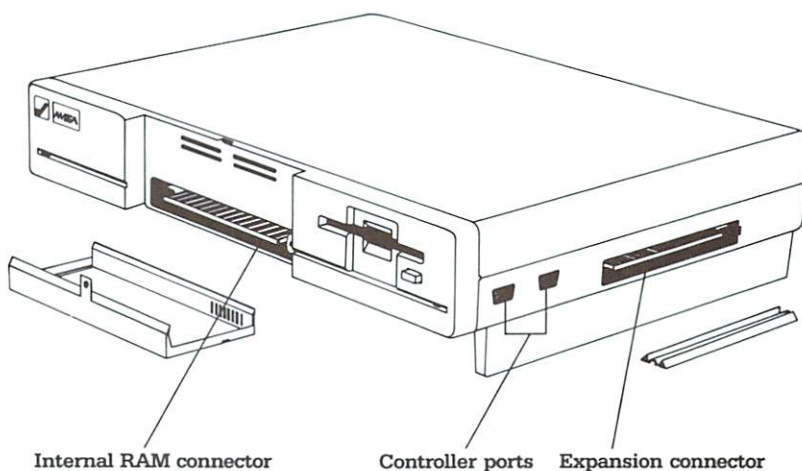


Figure 1-5.

The ports and connectors on the front and right side of the console.

The internal RAM connector

On the front of the console is a central panel that hides the internal RAM connector. If you grasp the panel firmly and pull it off, you can see the connector. You can use the connector to expand the Amiga's 256 kilobytes of internal RAM to 512 kilobytes to give it more memory to store programs and data. No soldering or tricky work is required: You simply buy a memory-expansion cartridge that plugs into the internal RAM connector.

The controller ports

On the right side of the Amiga, you can see two controller ports. The Amiga's mouse plugs into the first of these ports. The controller ports are not just for mice, though. You can plug in other controllers as well, such as joysticks, light pens, or touch tablets. The controller ports pass information from the controllers to the Paula chip so the Amiga can read the controller's position or actions. Two controller ports let you use two controllers at once.

The expansion connector

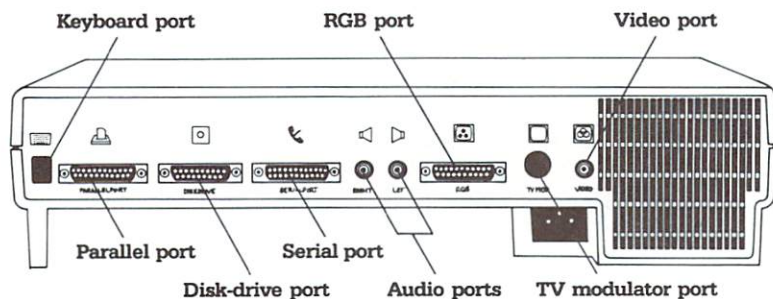
Just to the right of the controller ports is a hidden connector. You can pry off the small plastic panel covering it by using a penknife, a small screwdriver, or a long fingernail. Once you uncover it, you'll see the edge of the motherboard sticking out with what look like small copper teeth. This is the expansion connector. (Don't touch the teeth—a spark of static electricity from your finger could damage the Amiga.)

The expansion connector provides direct access to the chips inside the console. This connector makes the Amiga an "open machine" in the jargon of the computer market, because you can add components that require direct contact with the inner chips. The expansion connector lets you customize your Amiga.

Some examples of components you can add to the Amiga using the expansion connector are a hard disk drive to add more external data storage, extra RAM chips to add more memory (up to 8 megabytes, in addition to the maximum 512K internal memory), and a video-signal converter to send video images directly into the Amiga from sources such as video cameras. Most expansion components have a built-in connector identical to the Amiga's, so you can add more than one device to the Amiga without "using up" the expansion connector. (If you took the plastic cover off to look at the expansion connector, be sure to put it back on. It protects the connector, and ensures that you don't accidentally short out the inside of the Amiga.)

Figure 1-6.

The ports and connectors on the back of the console.



The keyboard port

If you turn the Amiga console around so you can see its back, you'll see a row of ports, shown in Figure 1-6. The first port on the left is the keyboard port, which passes information from the keyboard to the inside of the console. The keyboard cord plugs in here. If you run the keyboard cord under the console, you can slide the keyboard under the console whenever you're not using the machine.

The parallel port

Immediately to the right of the keyboard port is the parallel port. It sends data out of the console to whatever is attached to the port (usually a printer). Most parallel printers use a standard cable and data-transmission method called "Centronics parallel" after a series of printers sold by the Centronics Corporation. The Amiga's parallel port uses a modified Centronics standard, so with a special Amiga cable you can attach any printer that uses the Centronics standard.

The disk-drive port

To the right of the parallel port is the disk-drive port. You can connect up to three external disk drives here to add to the storage capacity of the system. To add more than one drive to this port, you connect each new drive to the back of the last one to create a chain. Every drive on the chain has direct memory access to the Amiga's RAM for fast data transfer.

The serial port

To the right of the disk-drive port is the serial port. The serial port is used to connect devices that need serial data from the computer, such as a modem (a device that transfers data over telephone lines between the Amiga and another computer) or a serial printer. There is a standard for serial-data transmission called "RS232" that determines the way the wires in a connecting cable are used and the way the bits are sent over the wires. The Amiga's serial port uses a modified RS232 standard, so with a special Amiga cable you can attach any serial peripheral that uses the RS232 standard.

The audio ports

The two audio ports are to the right of the serial port. They send out the audio signals that create the Amiga's sound. There is one audio port for the left channel and one for the right channel.

For good sound production, you can use the audio ports to connect the Amiga to a stereo system, just as you would connect a cassette deck or a compact disc player. The audio ports have a

line-level output (the same kind that cassette decks produce), which means the audio signals produced from these ports must be amplified in order to be heard. Using standard audio cables, you can connect these audio ports to a stereo amplifier, using the inputs for a tape deck, compact disc player, or an auxiliary device. If you want to record music you create with the Amiga, you can connect the audio ports to the inputs of a tape recorder to put your sounds on tape.

The RGB port

The port immediately to the right of the audio ports is the RGB port. If you use an RGB monitor for your computer display, you plug it into this port. The RGB port works with both analog and digital RGB monitors, although you need a special adapter cable to connect a digital RGB monitor.

The TV modulator port

To the right of the RGB port is the TV modulator port. This port sends out a standard television broadcast signal over a cable (included in the TV modulator package available from your Amiga dealer) you attach to the antenna inputs of a TV set. If you decide to use a TV set as your computer monitor, you connect it to the TV modulator port.

The video port

The video port is the port farthest to the right of the back of the console. It sends out an NTSC signal for a composite video monitor, the same type of signal that comes from the VIDEO OUT jack on the back of most video cassette recorders. (NTSC stands for the National Television Systems Committee, which sets the standard for video signals.)

If you use a composite video monitor for your computer display, you plug it into the video port. If you use a monitor that plugs into another port (a TV or RGB monitor), you can plug a video cassette recorder into the video port and record the images you view on the monitor.

COMMON PERIPHERALS

You can expand the minimum Amiga system with peripherals that add power and make the Amiga more useful. These common peripherals help save working time, and can transfer the results of the Amiga's work onto paper or over the phone lines.

One of the most useful additions to the Amiga system is a printer. Printers come in all shapes, sizes, and capabilities, and the Amiga will work with the majority of them. Most printers are used just to print text, but many will also print pictures from the Amiga's screen. These graphics printers are a very useful tool for artists.

Letter-quality printers

Businesses use letter-quality printers to create letters that look like they were done on a typewriter. The letter-quality printer puts characters on paper by striking an inked ribbon with fully formed characters on the ends of spokes, much like a typewriter. Although their printing is crisp and clear, letter-quality printers are of little use for graphics, since they can only create characters and not pictures.

Impact dot-matrix printers

The impact dot-matrix printer is the most common printer used with computers. It uses a print head with a vertical row of tiny pins that create characters by striking in different combinations against an inked ribbon as the head moves across paper. If you look closely at the characters, you can see they're made of dots imprinted by the pins. The printer creates pictures using the same method of building images with tiny dots. Most impact dot-matrix printers are strictly black and white, but some use colored ink ribbons to create color pictures.

Thermal-transfer printers

Thermal-transfer printers work much like dot-matrix printers, but use points of heat to create dots directly on special paper, similar to the dots created by a dot-matrix printer. Thermal printers also print pictures that look much like an impact dot-matrix picture; some even use a ribbon containing colored wax to create color pictures. Thermal-transfer graphics aren't usually as good as dot-matrix graphics—the placement of the dots on the paper isn't always as accurate and the dots are sometimes smeared—but thermal printers are usually much less expensive than other types.

Ink-jet printers

Ink-jet printers spray small drops of ink on paper to form text characters and graphics. Most models print only one color, but more expensive models are available that can print multiple colors using separate jets for each color. They're worth the cost if you want high-quality printed graphics, because they print with vivid and even coloration.

Laser printers

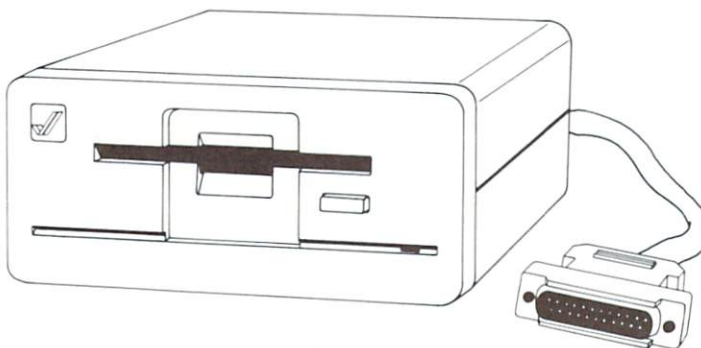
The laser printer is a type of printer that's out of the price range of most personal-computer users. It creates pictures and text using a photosensitive drum and toner the way a photocopier does, but creates the images on the drum with a laser instead of exposing the drum to a printed document through a lens, like a photocopier. Laser printers are very fast and quiet, and laser graphics are crisp and clear. Most laser printers print only in one color. You can vary the colors by using color-toner cartridges, but the text or graphics on each page will still be printed in one color, unless each page is sent through the printer again and reprinted using a different color toner. As laser technology advances, look for laser printers that will print multiple colors on the same page simultaneously. Prices should drop as well, until laser printers are more affordable for Amiga artists.

External disk drives

You can add up to three floppy-disk drives to the Amiga, using the disk-drive port in the back of the console. You can use either 3½-inch double-sided drives (shown below in Figure 1-7) or special IBM-compatible 5¼-inch drives designed for the Amiga (which must be used with the Transformer IBM emulator, available from your Amiga dealer), or combinations of both. The 5¼-inch drives use larger disks that don't store as much data as the Amiga's normal 3½-inch disks, but they can read IBM PC disks so you can use IBM software or read data created by an IBM PC.

Figure 1-7.

An external 3½-inch Amiga disk drive.



Each 3½-inch external drive gives you 876 more kilobytes of disk-storage capacity, enough by itself to store all the text in this book. A good reason to add an additional disk drive is that it makes it much easier to copy information from one disk to another.

When you use just the internal disk drive to copy files from one disk to another, you have to swap disks back and forth until the copy is complete. A second disk drive eliminates disk swapping, saving you time and aggravation.

Real gluttons for disk-storage space can use the expansion connector to add a hard-disk drive, which is capable of storing megabytes of information. Although this much extra storage space is generally used to keep a large number of records for database programs, it is also useful for animators and musicians who want to store long animation sequences or scores.

A tremendous data-storage capacity isn't the only benefit of a hard-disk drive: It also transfers information to the Amiga and back much faster than a floppy-disk drive. This speed can be very useful when you need to feed long sequences of animation images to the Amiga quickly.

External RAM

The Amiga can have a maximum of 512 kilobytes of internal RAM. If you really want to expand the Amiga's memory, you can add up to eight megabytes of external RAM by plugging external RAM cards into the external connector on the right side of the Amiga console. Since pictures and sounds require large amounts of memory, you might want the extra RAM to accommodate involved animation sequences or long musical scores.

Additional peripherals

There is a multitude of other peripherals designed to help you with specific tasks. A modem will connect the Amiga to phone lines so it can communicate with other computers and computer networks. Touch tablets and light pens help draw figures on the monitor screen without using the mouse. Synthesizers add to the Amiga's already powerful musical abilities, and video cameras, video recorders, and other various video accessories help bring outside video images to the Amiga, and bring Amiga images to the outside world. You can read more about additional peripherals in later chapters.

SOFTWARE

Since software exists only as electrical states in the Amiga's memory or magnetic charges on a disk, it's not as easy to examine as hardware. If you want to analyze its structure, you can look at printouts of the code the programmer used to create the software.

But that's a process not unlike reading through old federal income-tax forms, and isn't really much fun. (My apologies to die-hard hackers and accountants.) Practically speaking, software is what it does, so you can use a more enjoyable method of examining software by examining its results.

SYSTEM SOFTWARE

The software included with the Amiga, called the system software, runs the Amiga. Some of this software is stored inside the console on ROM chips. The rest is included on two floppy disks: the Workbench disk and the Kickstart disk. The system software has five main programs and many smaller routines that are structured in layers so they can work with each other.

An analogy helps to see how the system software works. Think of a factory set up to manufacture kazoos. The factory has machines for stamping out metal shapes, bending metal, attaching buzzing membranes, painting metal, and boxing up finished kazoos. The machines are run by a factory work force, men and women expert at making sure the machines turn out the finest kazoo craftsmanship. The individual workers are guided by supervisors who make sure things are running smoothly. Directing the supervisors are managers with expertise in each area of kazoo manufacturing—one manager for metal cutting, another for creative painting, and so forth. One step above the managers are directors—one for each different area, like shipping and quality control. A level above the directors are the vice presidents, who carry out the orders of the president of the kazoo company.

Whenever the president decides to do something new, like introduce a new line of Luciano Pavarotti-shaped kazoos, he tells one of his vice presidents to start production. The VP gives orders to the directors who need to be involved, who in turn issue their orders to the managers. The managers tell the work force what changes have to be made in running the machinery.

Although the Amiga doesn't manufacture kazoos, it runs like the kazoo factory. The hardware (the factory work force) does the nitty-gritty work, and the software (the management) runs the hardware. At the lowest level of the software is a program called Exec (the supervisors) that directly runs the hardware. Giving orders to Exec are small subprograms called libraries and devices (the managers) that specialize in areas like graphics, sound production, and math. At the level above the libraries and devices are AmigaDOS and Intuition (the directors), two programs that coordinate the work of

the software below to perform such tasks as moving data back and forth between the disk drive and the motherboard and presenting data to you in a way that's easy to understand. At the top level of the system software are two more programs, Workbench and CLI (the vice presidents). They take orders directly from you (the president) and pass them down to the rest of the system software. Figure 1-8 shows you the different layers of the Amiga system software.

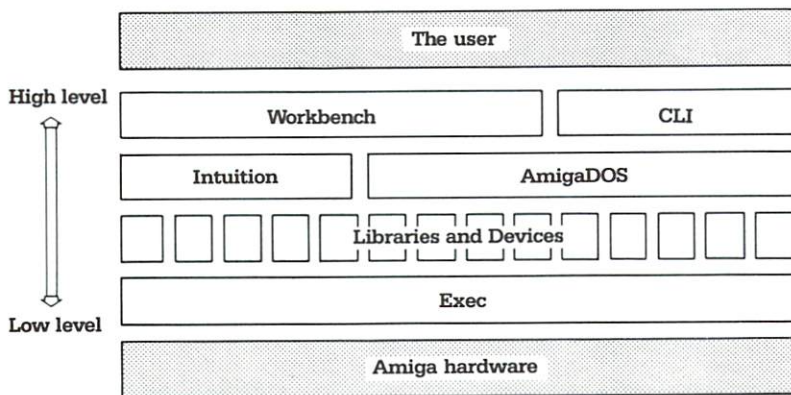


Figure 1-8.

The layers of system software between you and the hardware.

Software layering makes it easy for programmers to write Amiga programs. Programmers can use the lower-level programs built into the system software to do much of the work for them. The low-level programs like the devices and libraries include routines that perform tasks commonly used by all programs, such as pulling in data from the disk drive or reading what the user types on the keyboard. The programmer can simply ask the device or library to perform the task, instead of writing new code to do the same thing. For example, a programmer writing a new program doesn't have to write code to read the actions of the mouse. Instead, the new program can ask a lower-level program to read the mouse actions and return the information to the new program. This saves a lot of time and trouble for the programmer.

Exec

Exec (short for system executive) is the lowest-level program of the Amiga's system software. It's stored in the ROM chips inside the console and on the Kickstart disk. When you turn on the computer, it's Exec that starts things rolling. It first clears the Amiga's memory, runs a short test of the hardware, and then asks you to insert the Kickstart disk in the disk drive so it can bring the

rest of Exec into the Amiga's memory. When the Kickstart disk is finished loading, Exec asks for the Workbench disk so it can load the other system software.

Exec takes care of the fundamental hardware activities. It provides routines to pass data back and forth from chip to chip and to transfer data through the ports and connectors. Exec also has routines to check the keys on the keyboard, to monitor the actions of the mouse, to create video and audio signals, and to perform other tasks fundamental to the Amiga's operation.

Exec can run many different activities at once. For example, it can send characters out to a printer at the same time as it moves figures on the monitor screen. This ability is called multitasking, and is one of the features that makes the Amiga a powerful computer.

To juggle the separate activities of multitasking, Exec has to make sure that each activity shares the Amiga's chips without interfering with other activities. Exec is a little like a lawyer working for clients with conflicting interests: It spends a little time with each activity so that the activities don't think they're being ignored, and keeps each activity ignorant of the others.

Libraries and devices

Libraries are small programs that run just above Exec. Each library has its own specialty, and together the libraries cover a wide range of useful functions such as graphics, animation, mathematical operations, and putting text on the screen. Each of these libraries is available to higher-level software. For example, a program that creates video pictures can use the routines built into the graphics library, and a program that makes music can use the sound libraries, saving the programmer a lot of time and repetitious programming.

On the Amiga, devices aren't hardware, although their name would imply this. They're software, much like libraries, but they usually specialize in controlling a particular part of the Amiga's internal hardware, instead of providing more general functions like the libraries do. Devices take care of translating the keys pressed on the keyboard, tracking the movements and button clicks of the mouse, sending out data through the serial and parallel ports, and other hardware chores. Devices also generate the Amiga's sounds and keep track of time.

Advanced programmers in most languages can use the Amiga's libraries and devices directly to help them with their programs. Microsoft BASIC for the Amiga, for example, offers special commands that call directly on the Amiga's libraries and devices to perform tasks that BASIC itself might not be able to do.

AmigaDOS

On the level above the libraries and devices is AmigaDOS, short for Amiga Disk Operating System. Like a traffic dispatcher who keeps track of trucks on a highway, AmigaDOS keeps track of data and programs in the Amiga's memory and on disks in the disk drives. These programs and groups of related data are called files. When AmigaDOS stores files on a disk, it keeps track of each file's location on the disk, the size of the file, and when it was stored. When it comes time to bring the file back into RAM, AmigaDOS knows where to find the file and how to transfer it back to RAM.

AmigaDOS is also in charge of starting different programs. Since Exec can handle many different tasks at once, AmigaDOS can take advantage of Exec's multitasking ability to run different programs at the same time if you ask it to. In case of any conflict between programs, AmigaDOS knows which program is more important, and makes sure it gets chip time or storage space from Exec before the other programs do.

You use AmigaDOS whenever you load a program from disk and start it running. You can give commands to AmigaDOS using one of two higher-level programs: Workbench or CLI (discussed soon).

Intuition

Intuition is a program that works just above the libraries and devices, at the same level as AmigaDOS. It provides user-interface routines. The user interface, simply put, is the means you use to control a program. Intuition's routines create the elements of the Amiga user interface: screens, windows, menus, gadgets, and requesters. Intuition also provides routines that use libraries and devices to put text and graphics on the screen and read the mouse and keyboard to see what the user wants.

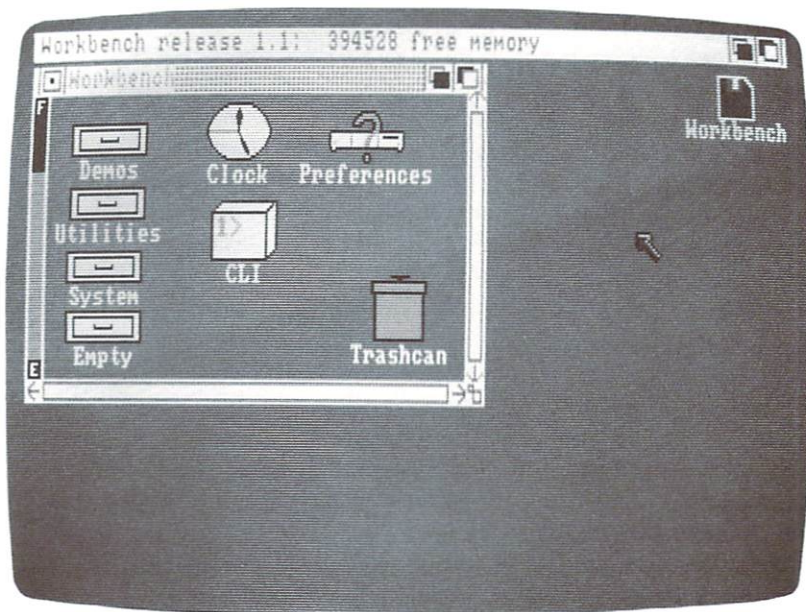
Programmers can use Intuition's routines to create an Amiga user interface for their programs. Instead of making their own menus, windows, and other features from scratch, programmers can ask Intuition to do it for them (for example, you'll learn how to make a window in an Amiga BASIC program in Chapter 4). Intuition's features encourage all programmers to use a consistent user interface so users won't have to learn a new set of commands and rules each time they learn a new program.

Workbench

Workbench is a graphics-based program that sits at the top level of the Amiga's system software and takes commands directly from you. It acts as an interpreter between you and AmigaDOS so you can use AmigaDOS to manage files on disks and in RAM. The display you see in Figure 1-9 (on the next page) is an example of Workbench.

Figure 1-9.

The Workbench display.



Workbench uses Intuition's routines to interpret your actions. When you double-click the mouse to open up a disk icon, Workbench uses Intuition to open a window on the screen and fill it with icons for the files on the disk. It asks AmigaDOS to look on the disk to see what files are there. Typical Workbench activities include copying disks, running programs, copying files, and erasing files from a disk.

CLI

CLI, short for Command Line Interpreter, is a text-based alternative to the graphics-based Workbench. Unlike Workbench, it doesn't make use of the Intuition user-interface routines to interpret your commands to AmigaDOS. Instead, it uses an older user interface that requires the user to type commands in a command line. CLI interprets each of these commands as an AmigaDOS activity and starts the activity the user typed in. Figure 1-10 shows an example of a session with AmigaDOS using CLI.

Although CLI is not always as easy to use as Workbench, it's more powerful. It provides more AmigaDOS activities than Workbench, and is particularly useful for advanced Amiga users. Examples of CLI activities not found in Workbench are setting a program to run immediately when you turn on the computer, and searching through files to find a particular piece of data. In addition, you can use CLI to run a series of AmigaDOS commands to create AmigaDOS programs to perform long and involved file-handling tasks.

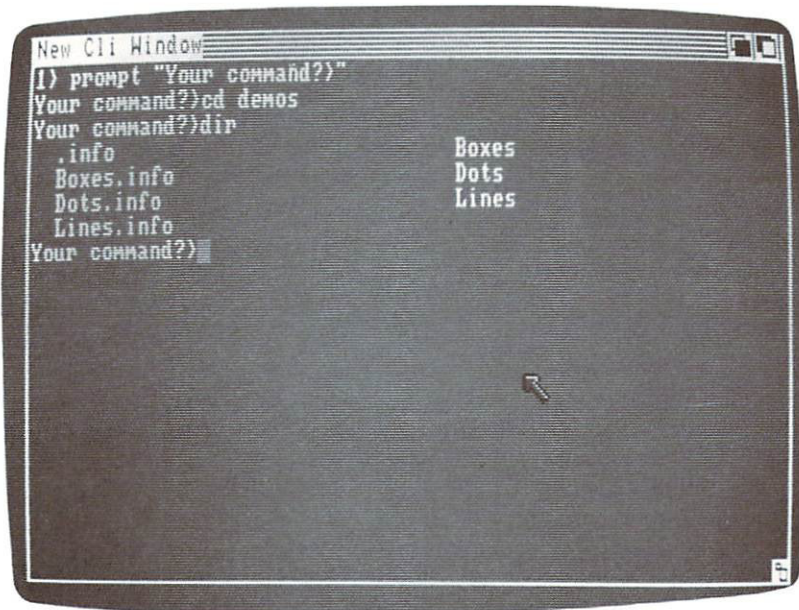


Figure 1-10.

Using AmigaDOS with CLI commands.

You won't find directions for using CLI in the "Introduction to Amiga" manual. You'll find CLI explained in the *AmigaDOS User's Manual*, which is available through your Amiga dealer or a computer book store.

SOFTWARE TOOLS

The system software is only the beginning of the software you'll use on the Amiga. For really practical results, you'll use software tools like Deluxe Paint, Deluxe Music, Deluxe Video, and Amiga BASIC, featured in the other sections of this book. These programs apply specifically to jobs you want to get done, and use the Amiga system software to get the results you want.

You've now learned about the Amiga computer system inside and out. The components you've read about in this chapter, both hardware and software, work together to make the Amiga the amazing computer that it is. But a computer is only a tool—however powerful—and in the long run it's you, practicing with the machine and exploring its potential, who does the work and creates the images, sounds, and animation that show off the Amiga's special talents.

SECTION 2

In this section, you'll learn how to create images with the Amiga computer. In Chapter 2, you'll learn the fundamentals of video graphics and in Chapter 3 you'll work with the graphics program Deluxe Paint. You'll also take a look at the other graphics programs for the Amiga, survey additional graphics equipment you can add to your Amiga, and in Chapters 4, 5, and 6 you'll work with the Amiga BASIC commands. When you're finished, you should be well on the way to becoming an accomplished Amiga artist.

Images





**CHAPTER TWO
A VIDEO GRAPHICS
PRIMER**

Most of us see video graphics every day, in forms such as television commercials or advertising displays in shopping malls. But despite their omnipresence, few of us understand how they're created. Unlike simpler media such as watercolors or pen and ink, video uses complex equipment to create images. Still, like any other medium, the quality of video images is determined by the quirks and foibles of its constituent materials. Just as the texture of paper and the transparency of watercolor paints give watercolor pictures an unmistakable quality, the glowing phosphors of a monitor screen give video images properties not found in any other media.

To do your best work creating images on the monitor, you should get to know the intricacies of video and learn what the Amiga is capable of creating with the video-handling graphics routines in its system software. In the chapters following this one, you'll learn how to apply the Amiga's graphic powers directly, by drawing and moving objects on the screen. But first, you should learn about the two crucial components that determine the way you see video images: the monitor that displays the image, and the eye that sees the image.

HOW YOU SEE BRIGHTNESS AND COLORS

Human eyes are complicated organs. They perceive shape, distance, brightness, and color, all by sensing the light coming in through the pupils at the front of the eye. Although seeing shapes and judging distance are obviously very important to our everyday lives, brightness and color are often more important to the artist. For video images, brightness and color are even more important—and the monitor you use with your Amiga has been designed to create a video image that takes advantage of the way the human eye sees colors and brightness.

PRIMARY COLORS

When you look at an image on a color monitor, you're actually seeing an illusion of color. In fact, the monitor can display only three primary colors: red, green, and blue. What the Amiga does is mix and match these primary colors, and the resulting combinations create the rainbow of colors you seem to see. This effect is directly related to the way your eye sees color.

At the back of the eyeball are two types of light receptors: rods and cones. The rods see a colorless world in shades of gray, while the cones see color. Although most people perceive a full range of

colors—red, yellow, green, cyan, blue, violet, and many colors in between—the cones in the eye can actually see only the three primary colors: red, green, and blue. The brain blends the varying amounts of red, green, and blue light coming into the eye to create additional colors.

The proportion of the primary colors blended together determines the resultant color. For example, you see yellow as a combination of equal parts of red and green light. If there is more red light than green light, the yellow takes on an orangish cast. If there's more green light than red light, the yellow becomes more of a greenish yellow. By varying the amounts of red, green, and blue light entering the eye, you can create a vast assortment of other colors.

RGB COLOR CREATION

To create different colors using graphics programs on the Amiga, you can adjust the program's color controls to vary the amount of red, green, and blue blended together to create a resultant color. This is called the RGB method of color creation. For example, if you open the Preferences screen from the Workbench, you're presented with three "slider" adjustments, one each for red, green, and blue. You create new colors using the RGB method by setting the three different sliders, which control the amount of each primary color. By setting the sliders shown in Figure 2-1 to different amounts, you change the ratio of red, green, and blue, creating an entirely new color.

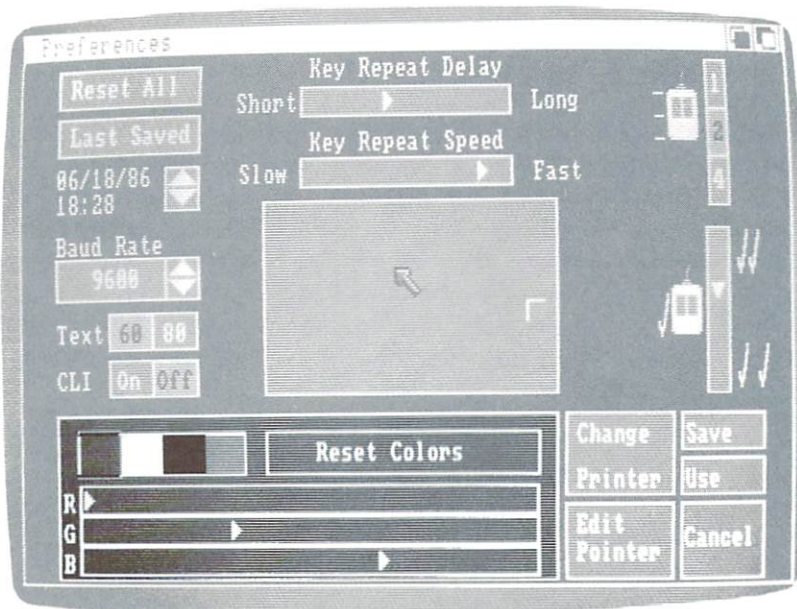


Figure 2-1.

The RGB controls of the Preferences program. Moving the sliders left diminishes the primary colors; moving them right increases the primary colors.

HIS COLOR CREATION

Although it's easy to set red, green, and blue blends to create new colors, it's not always easy to describe those colors as individual amounts of red, green, and blue, or to come up with exactly the color you wanted to create. People tend to describe colors using terms like "light purple" or "dark red" rather than descriptions like "eight parts red, eight parts blue, and three parts green." There is another way to describe colors using three attributes: hue, intensity, and saturation (HIS). These attributes describe colors more like the way we see them.

Hue

Hue is the most apparent color attribute. You usually talk about hues when you talk about color, using words like red, purple, orange, and green. Hue is set by the mixture of primary colors in a color, and is changed by the ratio of these primary colors. For example, a mixture of blue and green light where the blue light is twice as strong as the green light will create a hue described as greenish blue. A mixture of blue and green light where the green light is twice as strong as the blue light creates a hue described as bluish green. Since hues are often mixtures of other hues, many of them have names like reddish orange, purplish pink, and yellowish brown.

Intensity

The overall strength of light determines the intensity of a color. Intensity can run the range from black (no light at all) to radiant (maximum light strength). As an example of different color intensities, think of the difference between the colors at the beginning of a sunrise and the colors when the sunrise is full. Many of the hues are the same, but their intensity grows stronger as the sunrise progresses, so the colors become much more radiant.

Hue and intensity are separate attributes—one can change without altering the other. For example, a greenish-blue hue is created with a mixture of two parts blue light and one part green, so the ratio is two to one. If the strength of both the blue and green light is doubled so the new mixture is four parts blue light and two parts green, the ratio is still two to one, so the hue is still greenish blue. The intensity changes, though—the second color mixture is much more radiant.

Saturation

The saturation of a color changes as more white is mixed in with it. Fully saturated colors have no white mixed in, and so look more vivid. Colors with little saturation are diluted by white and look pale, like pastel colors.

Since radiant white is actually an equal mixture of red, green, and blue, when you add white to a color you are actually adding red, green, and blue in equal parts. Therefore, saturation is tied together with hue: Both hue and saturation change as white is added, since the equal parts of red, green, and blue used to create white change the ratio of primary colors in the resultant color. Whether a color changes hue or saturation is mostly a subjective judgment—does the new color look paler, or does it actually look like a new hue?

Some Amiga graphics programs use HIS to create new colors. For example, Deluxe Paint, described in the next chapter, uses three sliders to control the hue, intensity, and saturation of a color, just as the Preferences program uses RGB sliders. Deluxe Paint takes your HIS settings and translates them into RGB settings that the Amiga needs to create the colors on the monitor screen.

HOW A MONITOR CREATES IMAGES

To many people, the inner workings of a computer monitor or television set are every bit as mysterious as the biochemical processes of the human eye. Monitors need not be quite so mysterious: They use principles of electronics that have been around for years.

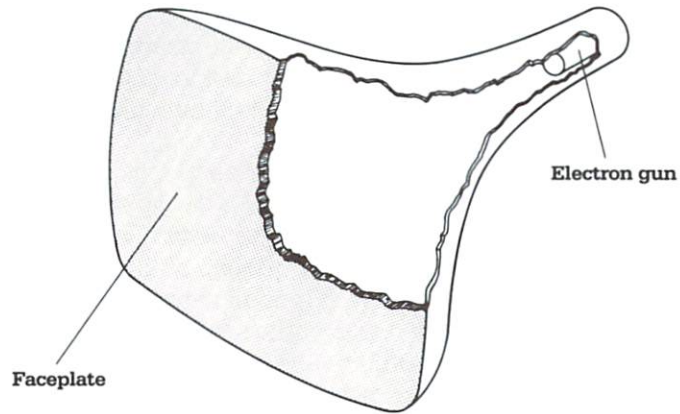
To learn about monitors, it's easiest to first learn about monochrome monitors, which display pictures using only one color. Although you probably won't use a monochrome monitor with your Amiga, understanding the principles of a simple monochrome monitor makes it much easier to understand the color monitor you most likely use with your Amiga.

MONOCHROME MONITORS

The heart of a monochrome monitor, or almost any monitor for that matter, is the cathode-ray tube (CRT for short). The CRT is a large vacuum tube with a rectangular faceplate for displaying images, and an elongated stem containing an electron gun protruding from the rear (see Figure 2-2 on the next page).

Figure 2-2.

Cutaway view of a cathode-ray tube.



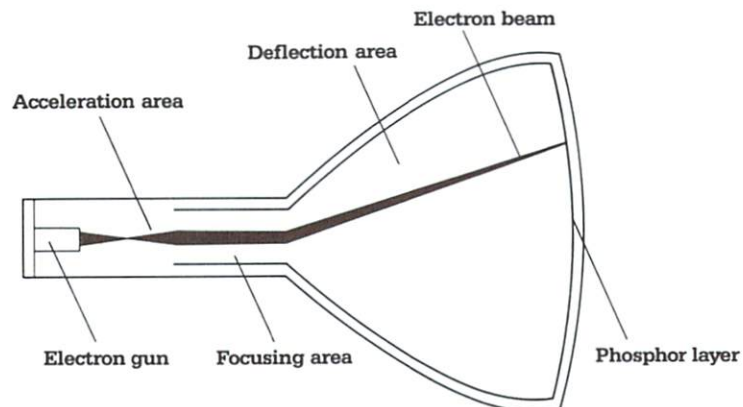
When you turn on the monitor and let the CRT warm up, the electron gun fires a stream of electrons at the front of the tube, which is covered with a layer of phosphor particles. Each phosphor particle glows briefly when it's hit by an electron, and remains dark when it's not. A black-and-white television, which is the most common monochrome monitor, uses phosphor that glows white. Some monochrome computer monitors use green or amber phosphor instead of white phosphor.

The electron gun

To create an image on the faceplate, the CRT focuses the stream of electrons fired by the electron gun into a narrow beam and then aims it at the phosphor particles. Figure 2-3 shows how the CRT does this. First, it uses magnetic fields to accelerate the electrons to the speed necessary to make the phosphors glow when they are struck, and then it uses other magnetic fields to focus the electrons into a narrow beam precise enough to hit just one small spot on the phosphor layer at a time. Finally, magnetic fields in the deflection area pull the electron beam up or down and right or left to aim it at different spots on the faceplate.

Figure 2-3.

Focusing and aiming the beam of a CRT electron gun.



Raster scanning

To create an image across the entire faceplate, most monitors use a technique called raster scanning. The area of the screen that will be struck by the electron beam is called a raster. The computer or the television circuitry controlling the monitor divides the raster into several hundred horizontal lines called raster lines. It then sends signals to the monitor that move the beam of the electron gun across the CRT's faceplate to draw each raster line.

To cover the entire faceplate, the electron beam starts in the upper left corner of the screen and scans from left to right across the top of the screen. The strength of the electron beam determines how brightly each phosphor particle glows as the beam scans over it. Where the phosphor should be dark, the electron gun fires no electrons. Where the phosphor should be fully lit, the electron gun fires electrons at full force. For shades in between, the electron gun fires at partial strength so the phosphor doesn't glow as brightly.

When it reaches the right end of the raster line, the electron beam sweeps back to the left side of the screen in a motion called the horizontal retrace. It then starts a new raster line just below the last raster line. When it finishes this raster line, it sweeps back and starts yet another raster line, continuing all the way to the bottom of the screen, displaying raster line after raster line. When it reaches the bottom it has finished its vertical sweep, and goes back to the top left corner of the screen in a motion called the vertical retrace. Figure 2-4 shows the motion of the electron beam in a typical raster scan.

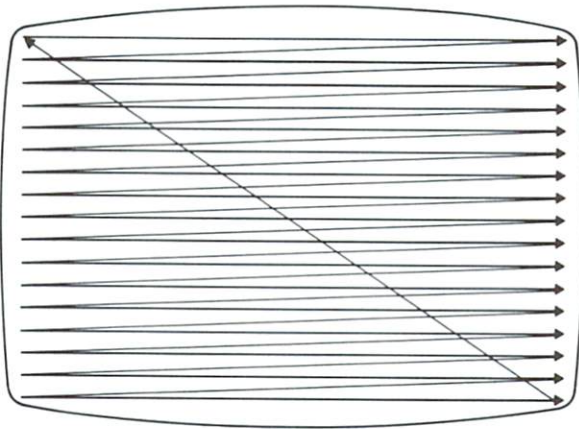


Figure 2-4.

The electron-beam pattern used in raster scanning.

The raster scan has to take place very quickly, because phosphor particles stop glowing very soon after the electron beam moves

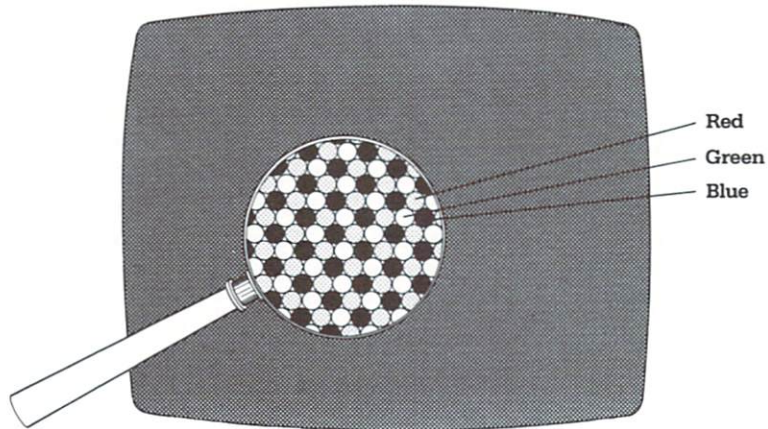
away. To keep all the phosphors glowing on the screen, the electron beam sweeps over the entire faceplate sixty times a second. This raster-scanning rate is called the refresh rate.

COLOR MONITORS

On the faceplate of a color monitor are three different colored phosphors: red, green, and blue, in clusters of three. If you turn on a color monitor and look at it under a magnifying glass, you can see the three different colored phosphor dots glowing, as shown in Figure 2-5.

Figure 2-5.

The phosphor dots on the faceplate of a color monitor.



At the back of the color CRT are three electron guns, one for red phosphors, one for green, and one for blue. A color mask located just behind the phosphor layer of the CRT has a grid of tiny holes, one for each cluster of colored phosphors. Each hole restricts the beams from the three electron guns as they sweep across the screen, so that each beam can strike only the appropriate colored phosphor in each cluster, as illustrated in Figure 2-6. You can see in the figure that the blue electron gun can hit only the blue phosphor in the phosphor cluster; the hole is too small to let the beam strike the red and green phosphors in the cluster. Likewise, the red gun can hit only the red phosphor, and the green gun can hit only the green phosphor.

A color monitor uses a raster scan just as a monochrome monitor does. The color picture is broken up into raster lines, and the three electron beams sweep in unison across the screen, working their way across and down sixty times a second. Instead

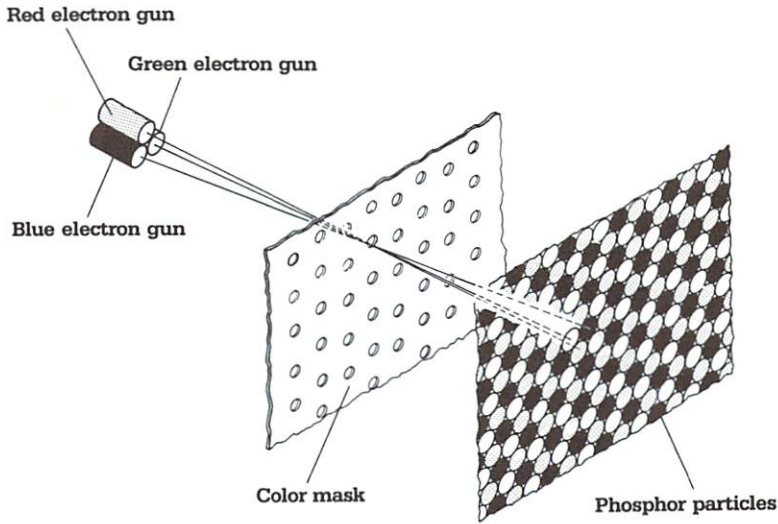


Figure 2-6.

The color mask in a color CRT restricts each of the three electron beams to its appropriately colored phosphor in a phosphor cluster.

of creating monochrome shades, the three electron guns vary their intensity to create different ratios of red, green, and blue in the phosphor clusters to create a variety of colors on the screen.

To control a color CRT, a computer or television tuner must send separate intensity signals for each electron gun and another signal to control the sweep of the electron beams. In an RGB monitor, these signals are sent through separate wires directly to the electron guns and the deflection grids. In a composite monitor or a color television set, the signals are mixed together, sent to the monitor over a single wire, then separated once again into separate signals to send to the electron guns and deflection areas. Combining and separating the signals deteriorates the picture quality. This is why RGB monitors offer a clearer picture than composite monitors or television sets.

TRANSFERRING A VIDEO IMAGE TO OTHER MEDIA

When you look at an image on the monitor, only you and whoever is able to peer over your shoulder can see the picture. At times you will no doubt want a larger audience for your graphics, and you will need to transfer them to other media that are both portable and accessible to more people, such as videotape, photographs, and printed copies.

Transferring video images to other media is not always simple. Resolution differs from medium to medium, the colors don't always match the way they should, and the sense of proportion isn't always the same. Basically, the quality of the image when translated to a different medium depends on the quality of the software you use to translate the image, the quality of the equipment you use to reproduce it, and your skill and knowledge of the medium used.

VIDEOTAPE

Recording the Amiga's video images on videotape is an easy way to store and reproduce Amiga graphics. The Amiga can send out a video signal from its composite video port directly to a video tape recorder instead of to a composite monitor. The Amiga has the capability of driving more than one monitor at a time, so you could display the graphics on an RGB monitor connected to the RGB port, while recording the same image on a VCR connected to the composite video port. The video recorder stores the images on videotape, so you can send the tape to other people with the same type of recorder, or broadcast the tape over a television transmitter if you're lucky enough to have access to one.

Videotape is an excellent way to store Amiga graphics. It can capture motion and accompanying sounds, and it plays everything back on a familiar medium: a monitor. There are some minor drawbacks. A computer-generated image on videotape usually doesn't look as good as the original image. Also, if your original image is a still image, the VCR can only display it until the tape runs out—you can't keep the image on the monitor indefinitely.

PHOTOGRAPHING THE MONITOR SCREEN

Another relatively simple way to reproduce an Amiga video image is to photograph the monitor screen. However, you can't just point a camera at the screen and shoot and expect to get good results. You must follow a few precautions to get effective photos of your graphics.

To photograph the monitor screen, you should be in a completely dark room so external lights won't reflect off the monitor screen and fade the picture. Because the standard raster scan takes a full 1/60 of a second, you should also be sure the shutter speed of the camera is slower than 1/30 of a second, which means that you'll need a tripod to steady your camera as you shoot. Faster shutter speeds will catch the raster scan of the monitor with its job only

partly finished. For example, a shutter speed of 1/120 of a second will catch the raster scan with only half the picture drawn on the monitor screen.

Prints made from your video photographs have an obvious advantage—they're portable. You can send them through the mail or mount them on the wall. They're also easy to look at—you don't need to take time to plug them in, turn them on, or load them. Like any other kind of photographic print, you can use them in thousands of different ways.

There are some disadvantages to color prints. They don't always reproduce the colors on your monitor with complete accuracy, and they don't glow the way your monitor does. One way to improve color reproduction is to shoot photographs using slides. Slides have the added advantage of looking radiant, like the monitor screen, when you project them on a projector screen.

Prints and slides have a common distortion problem, because a monitor screen usually has a curve to it. When you look at it directly, images don't seem curved because your brain adjusts for the curvature. When you photograph a video image, you transfer it to a flat medium, and the curvature becomes noticeable. The picture bulges a bit in the center, like a picture T-shirt on a man with a beer belly.

PRINTING VIDEO IMAGES

The most common way to reproduce an Amiga video image is to print it out on a printer that's capable of printing graphics. It's fast, simple, and inexpensive (once you've paid for the printer). You don't have to wait days for the photo finishers to return your photos. You can pick up the results in a matter of minutes, then slip them in an envelope and mail them off or post them on a bulletin board. Printed images also avoid the curvature problem you get in screen photographs.

The quality of printed images varies a lot from printer to printer. At its best, a printed color image looks very much like the video original. At its worst, a printed image will provide you with a good reason to give your printer away to the next used-computer-equipment drive by the local Boy Scouts.

In transferring a video image to paper, the most critical component affecting the fidelity of the reproduction is probably the quality of the printer itself. Another aspect of the problem, though, is converting the video image into data that the printer can understand and subsequently reproduce. This is where the quality of the software driving your printer is very important, because it has to overcome many different obstacles.

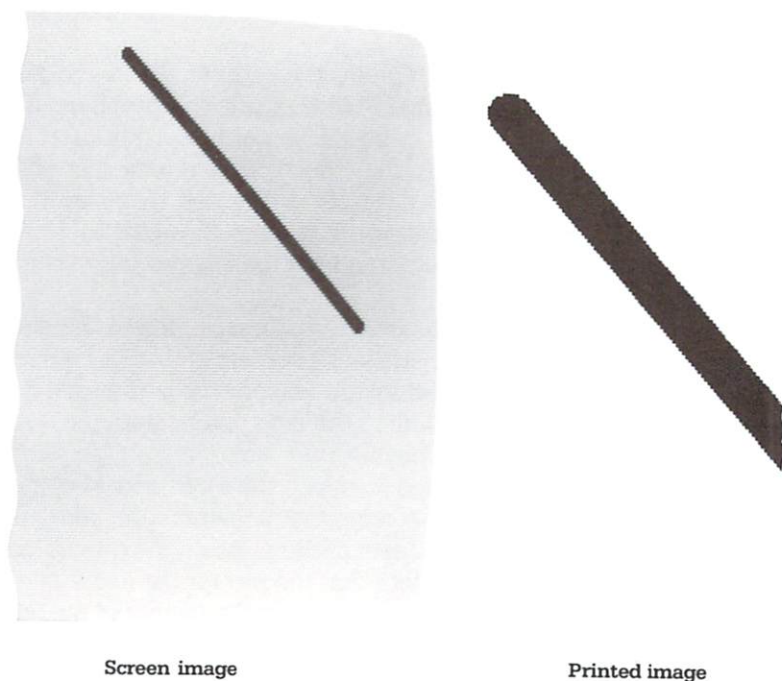
Resolution

One of the first problems in translating video to print is that most printers can print a picture with more detail, or higher resolution, than the computer can display on the monitor. An average dot-matrix printer can put over 900 dots of ink across the width and 1200 dots down the length of an 8½-by-11-inch sheet of paper. The Amiga's most finely detailed display on the monitor has 640 picture elements across the width of the screen and 400 picture elements down the length, a display of lower resolution than the 900-by-1200 printed picture. To print a picture from the monitor on a printer, the Amiga has to convert the image.

The simplest way to convert video resolution into print is to use blocks of ink dots to represent each video picture element (pixel for short). Each element in a video image that measures 320 by 200 pixels could be printed with a corresponding 3-by-3 block of ink dots, stretching the 320-by-200 image into 960-by-600 ink "blocks." The problem with converting video pixels into blocks of ink dots is that it makes the printed image look jagged. In Figure 2-7, you can see a diagonal line printed on a dot-matrix printer using blocks, and the line as it appears on a video monitor. The printed line looks pretty jagged.

Figure 2-7.

"Jaggies" created by printing out a video image.



High-quality printer software uses more than one printer dot to represent a screen pixel, but instead of using blocks, it uses a more intelligent translation scheme to smooth out the jaggies (computer jargon for the stairstep effect in curved and diagonal lines) in the printed translation of the screen image. This uses the printer's higher resolution to its best advantage, as you can see in Figure 2-8. Smoothing out the jaggies this way is called anti-aliasing. It can make images look a lot better, but the software has to second-guess the artist's intentions. If the artist intentionally puts jaggies in the picture, they might be smoothed out in a print made using anti-aliasing software.

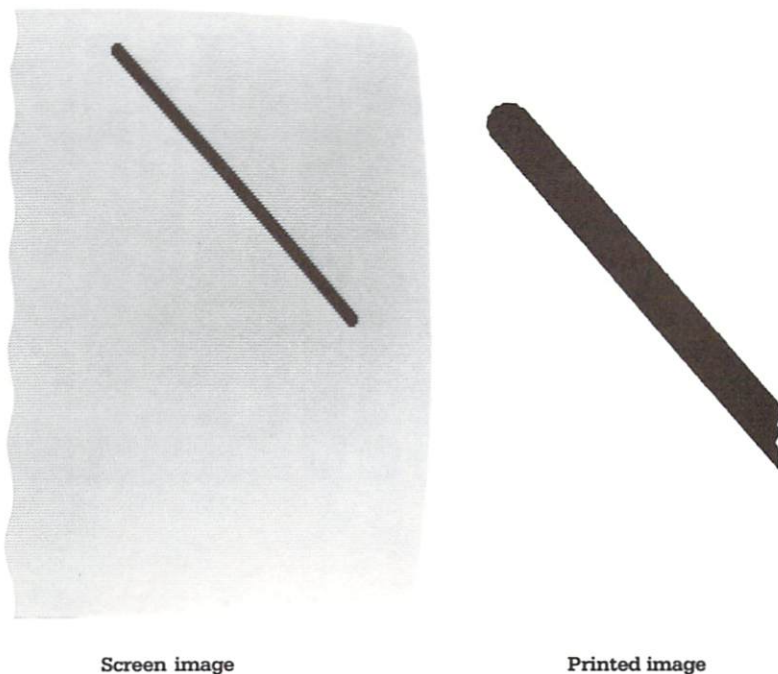


Figure 2-8.

Anti-aliasing smooths the jaggies in a printed diagonal line.

Converting color images to black-and-white printouts

Most computer printers don't print in color; they're strictly black and white. To translate the colors of an Amiga image into black-and-white ink dots for those printers, the printer software has two choices: It can print the image as strict black and white, like a silhouette, or it can print the image in shades of gray, which is known as a gray scale.

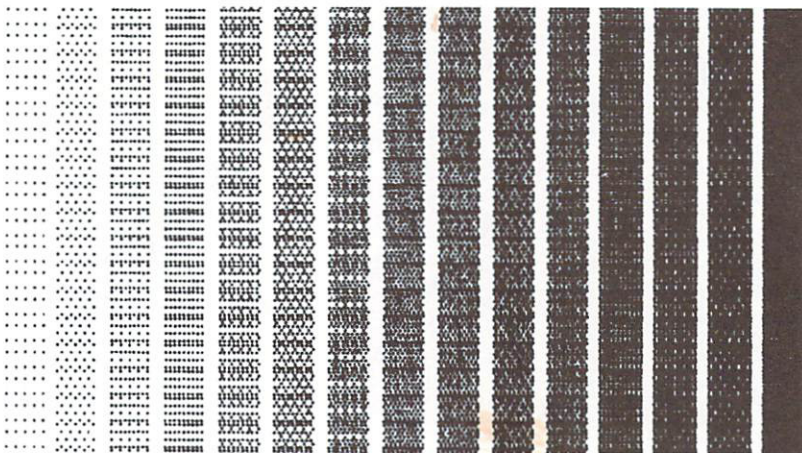
Printing the image in strict black and white is the easiest way to translate the video colors. The software simply sets a brightness reference point. Any pixels in the video image that are brighter than the reference point are printed as solid white, and any colors

that are darker are printed as solid black. The obvious disadvantage to strict black-and-white printing is that it loses all the subtlety of shade in the original image. It can be used to good advantage, though, to translate a color picture with strong contrast into a silhouette.

To turn colors into shades of gray, the printer software uses black dot patterns on the printer to create different shades of gray. Each shade of gray has a different dot pattern. Figure 2-9 shows some enlargements of gray dot patterns. To turn the video colors into gray, the software interprets bright pixels as lighter shades of gray and dark pixels as darker shades of gray, then prints them using the corresponding dot patterns.

Figure 2-9.

Shades of gray on a dot-matrix printout (enlarged).



Printing in color

Color printers don't have to interpret video colors as shades of gray, but they do have other tricky issues to resolve. Most important is that color printing uses a set of primary colors that aren't the same as video primary colors. Color printers use cyan, yellow, and magenta in different mixtures to create other colors, and often add black to give the picture more contrast.

Translating red, green, and blue directly into cyan, yellow, and magenta wouldn't be difficult, except for one important factor: The primary colors on a monitor screen can change in intensity but a printer's primary colors can't. The primary colors on a color printer have just one intensity because a dot of ink, unlike a phosphor, can't glow at different strengths. When a color printer mixes two

primary colors together, it gets just one resulting color. Because of this, most color printers have just six different solid colors available to them: cyan, yellow, magenta, green (cyan and yellow mixed), red (yellow and magenta mixed), and violet (cyan and magenta mixed).

To approximate the thousands of different colors the video screen can create, color printers use a process called dithering. Dithering is an odd-sounding term that means the printer overlays dots of one color on a solid field of another color. Viewed from a slight distance, dithering creates a new color. One example is a field of solid red with dots of black sprinkled throughout, resulting in dark red.

Dithering gives a color printer a wide variety of colors, but its color range is still limited compared to that of the monitor screen. As a result, an Amiga image transferred to a printer won't show subtle variations in color. The printed colors also have a textured look as a result of dithering that makes them look different from their video counterparts.

THE AMIGA'S UNIQUE GRAPHICS FEATURES

Now that you've seen how video images are created on the Amiga's monitor, seen by the human eye, and transferred to other media, you can look at the Amiga's specific graphics capabilities with a better understanding of its accomplishments. The Amiga was designed to be an exceptional graphics computer, and the power of its hardware, combined with the graphics routines in the system software, enable it to create video images of power and subtlety.

The following pages describe the graphics features available in any Amiga computer system running with its system software—the software on the Kickstart and Workbench disks you use to start the system. Although you may not be able to use all of these features directly (unless you're an advanced programmer capable of using the Exec libraries and devices), most of the features are available through application programs like Deluxe Paint, and through programming languages like Amiga BASIC, as we'll see in later chapters. As more software comes out for the Amiga, you can expect to see even more of these graphics features available in application programs.

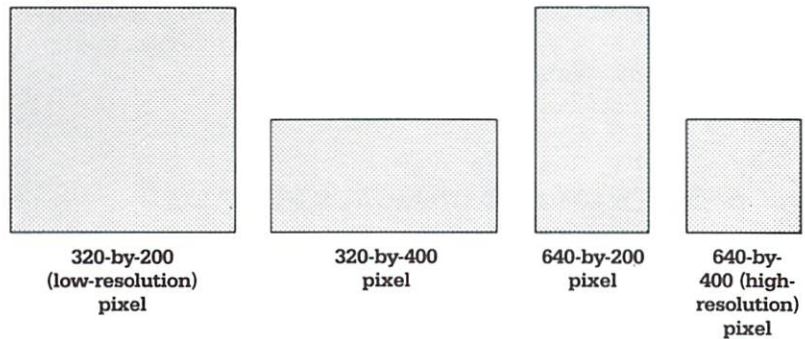
PICTURE DETAIL

Like most other microcomputers, the Amiga creates images on your monitor screen using pixels, those tiny boxy dots that give computer images a slightly jagged quality. Creating a picture with pixels is like building a house with bricks: Everything has square corners. The trick to rounding off the jaggies in the picture is to use smaller pixels; at a distance, you can't see all those square corners. The Amiga has a choice of four different-sized pixels for varying degrees of detail and smoothness. You can see these in Figure 2-10.

The size of the pixels you use in a screen determines the *resolution* of the screen. The smaller the pixels, the higher the resolution and the finer the detail in the pictures on the screen. The larger the pixels are, the lower the resolution is and the coarser the pictures are on the screen.

Figure 2-10.

The four different sizes of Amiga graphics pixels, greatly enlarged.



The lowest Amiga screen resolution is 320 by 200—that is, you can fit 320 pixels across the screen and 200 pixels from the top to the bottom of the screen. The picture in Figure 2-11 is displayed using the Amiga's low-resolution screen. If you look closely, you can see that the pixels are roughly square in shape. From a distance there is plenty of detail and the jaggies aren't apparent.

In two other Amiga screen resolutions, the low-resolution pixel is cut in half to create a rectangular pixel. In 320-by-400 resolution, the low-resolution pixel is cut in half horizontally to look like a brick lying flat. In 640-by-200 resolution, the low-resolution pixel is cut in half vertically to look like a brick standing on end. These skinny vertical pixels are useful for displaying text; by using them, you can fit more characters across the screen. In Figure 2-12, you can see both text and graphics displayed on an Amiga 640-by-200 resolution screen.



Figure 2-11.

A picture using the Amiga's 320-by-200 resolution screen. ("Venus" courtesy of Avril Harrison)

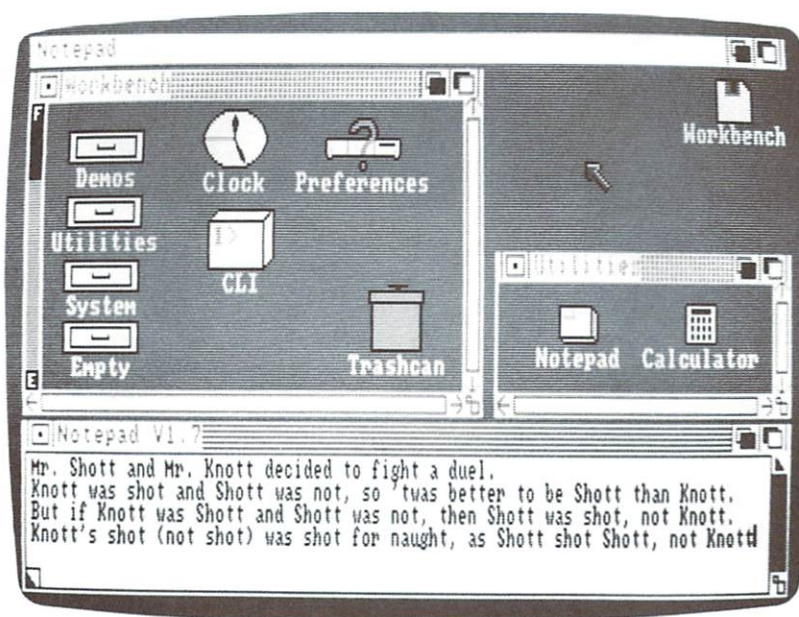


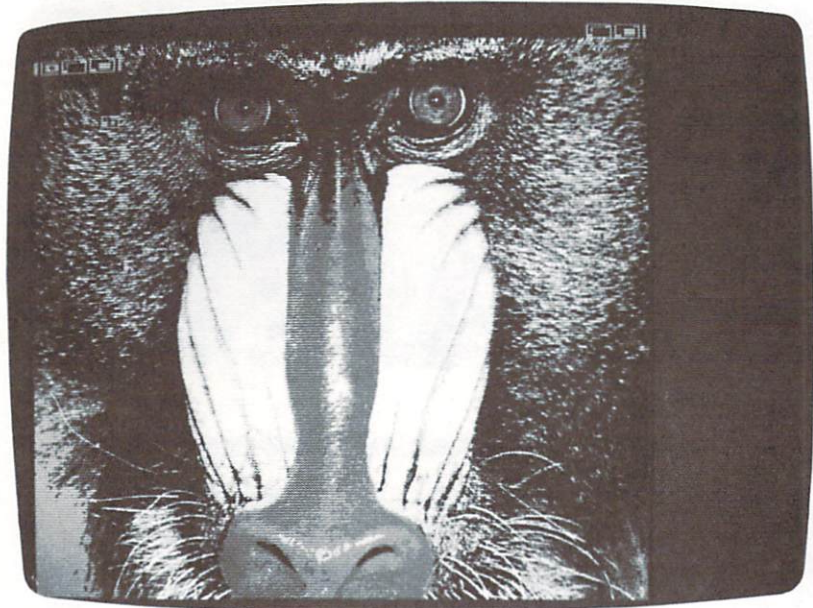
Figure 2-12.

A 640-by-200 Amiga screen: the Workbench display.

In the Amiga's highest-resolution screen, the pixel is a square shape just one-fourth the size of a low-resolution pixel. The high-resolution screen measures 640 pixels across by 400 pixels from top to bottom. With pixels this small, the detail of the picture is very fine, and it's hard to see jaggies. In Figure 2-13, you can see a high-resolution Amiga picture. It has enough detail to show the individual hairs on the mandrill's face.

Figure 2-13.

A picture using the Amiga's 640-by-400 resolution screen.



How resolution affects memory

Why are there different resolutions available on the Amiga? Wouldn't it be best to use the high-resolution screen for all purposes? The answer lies in the memory required for each resolution. The Amiga stores the information for a picture in its memory, and the higher resolution the picture uses, the more RAM is required to store it. For example, a low-resolution picture has only 64,000 pixels to store (320 times 200). And a high-resolution picture has 256,000 pixels to store (640 times 400), four times the number of pixels in a low-resolution picture.

With four different resolutions available, you can choose the resolution that best fits your needs without using up too much of your Amiga's RAM. You can also use the different-size pixels as

different graphic media to get fine or coarse effects, much like a painter picking different grades of paper for a watercolor. You'll learn how to specify the resolution you want in later chapters.

Mixing resolutions

If you would like to mix resolutions on the monitor, the Amiga can accommodate you by allowing you to divide the monitor display into horizontal areas called screens (not to be confused with the monitor screen itself), each with its own resolution. Intuition, the user interface, allows you to display different resolutions on the monitor simultaneously by layering many screens. You can drag screens down with the mouse pointer to reveal any screens underneath, and you can drag screens up to cover any screens that were underneath. Figure 2-14 shows layered screens of different resolutions on the monitor screen.

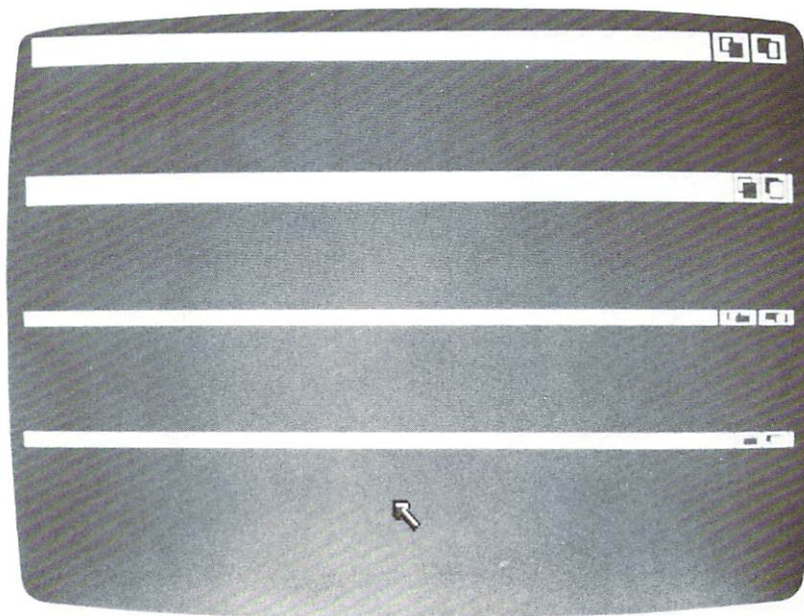


Figure 2-14.

An Amiga display with all four screen resolutions displayed simultaneously.

Most Amiga programs, like Workbench for example, use only one screen with a pre-set resolution. Some programs, like Deluxe Paint, offer you a single screen in the resolution of your choice. A few other programs, like Deluxe Video, use two or more screens with different resolutions that you can move up and down on the monitor. If you want to create your own screens, you have to use a programming language like Amiga BASIC or C. (You will learn more about screens and how to create them in Chapter 4.)

THE AMIGA COLOR PALETTE

Resolution is just one aspect of an Amiga picture; the other one is color. Resolution gives a picture height and breadth, color can give it depth.

The Amiga can create up to 4096 different colors on the monitor screen by combining red, green, and blue in varying amounts. It doesn't usually display all of these colors at once, though. Like increasing picture resolution, increasing the number of colors in a picture requires more memory to store the picture. To keep picture memory requirements down to a reasonable size, the Amiga usually limits the maximum number of colors on one screen to 32 (except on 320-by-400 and 640-by-400 resolution screens, where it sets a maximum of 16 colors). This provides a wide color range without using up so much memory that the Amiga can't run any programs.

The Amiga uses an ingenious system to allow a great deal of flexibility for those 32 colors. It colors its pictures like a color-by-number painting; each pixel in the picture is assigned a number from 0 to 31. Then the Amiga colors in each pixel with the color its number stands for.

The colors for each color number are stored in 32 separate color registers, also numbered from 0 to 31. These color registers are small, individual sections of memory that store color as mixtures of red, green, and blue in different proportions. The Amiga sends these mixtures to the monitor, where the monitor uses them to create different colors on the screen by matching each pixel's color number with the color in the corresponding color register. Each of the color registers can store one of the 4096 possible combinations of red, green, and blue that the Amiga is capable of producing, so you can choose from an extremely large palette to create virtually any combination of colors to display on the screen.

Although the principal advantage of color registers is that they provide a wide variety of colors without eating up a lot of RAM, they have another advantage. When you change the color in a single color register, every pixel on the screen with the same color number as that color register also changes color. You can use color registers this way to test different color combinations easily.

For example, if you use Deluxe Paint to design a fabric pattern on the Amiga that has little green worms on a purple background, you might want to try little green worms on an orange background instead. You don't have to go back to the pattern and take the time to fill in the background with orange; instead, you display the color palette on the screen, choose the background color, and change it

from purple to orange using the color-creation sliders on the side of the palette (you'll see how to do this in the next chapter). Every purple pixel on the screen turns to orange, and you can see the results immediately.

The Amiga can put all 4096 colors on the screen at one time using a special mode called Hold and Modify (known as HAM for short), which smears the colors horizontally to create very subtle shading. For example, the Amiga can use HAM to shade a round red vase to make it look three-dimensional. Where the vase curves out and catches the light, it would use a bright red. It would then subtly shade the red, turning it to darker shades where the vase curves into the shadow. This gives it a smooth, glowing appearance. Figure 2-15 shows the full shades of a display using HAM.



Figure 2-15.

An Amiga picture using the Hold and Modify (HAM) mode for shading. *(Image courtesy of NewTek and Mitchell Lopes)*

At the time of this writing, there are no graphics programs that use HAM, although there will probably be some in the future. There are some video digitizers for the Amiga that convert an image from a video camera or other video source into a HAM picture that uses subtle shading to reproduce the video image.

DRAWING PICTURES

The Amiga has special routines in its graphics library that help draw pictures on the screen. These graphics routines create the different components of any video picture: They can draw lines, fill in areas with a specific color or pattern, copy one section of a picture to another section, change colors, and perform other important graphics functions.

The graphics routines make use of a special section of the Agnus chip (one of the three custom chips) called the blitter. Blitter is short for "bit-mapped block transfer," a mouthful that means it quickly shuffles around large blocks of data in memory. When that data happens to be in the graphics-display section of the Amiga's memory, the blitter can draw figures very quickly so you don't have to wait a long time for a picture to appear on the monitor screen.

The graphics routines are used frequently in Amiga software. Any of the graphics commands in Amiga BASIC or the drawing functions in programs like Deluxe Paint use the graphics routines to accomplish their tasks. Workbench uses the routines to draw icons and windows on the screen.

CREATING TEXT

Libraries in the Amiga's system software also create characters so the Amiga can put text on the screen. They use a flexible system that resembles color registers, but instead of storing a red-green-blue color combination in each register, it stores the design for a character. The full set of character designs stored in memory—including all the characters in the alphabet, numerals, punctuation marks, mathematical symbols, and other special characters—is called a font.

As you type characters in at the keyboard, the Amiga uses the designs it has in memory for the font to create each character it puts on the screen. If you change a font, the Amiga changes the designs in all the registers, and any new characters appearing on the screen use the style of the new font. One font might be blocky and straight, another flowing and elegant. For example, in a word-processing program, you can have the Amiga use one font to display a headline, and use a second font to print a personal message. Figure 2-16 shows some of the fonts the Amiga uses.

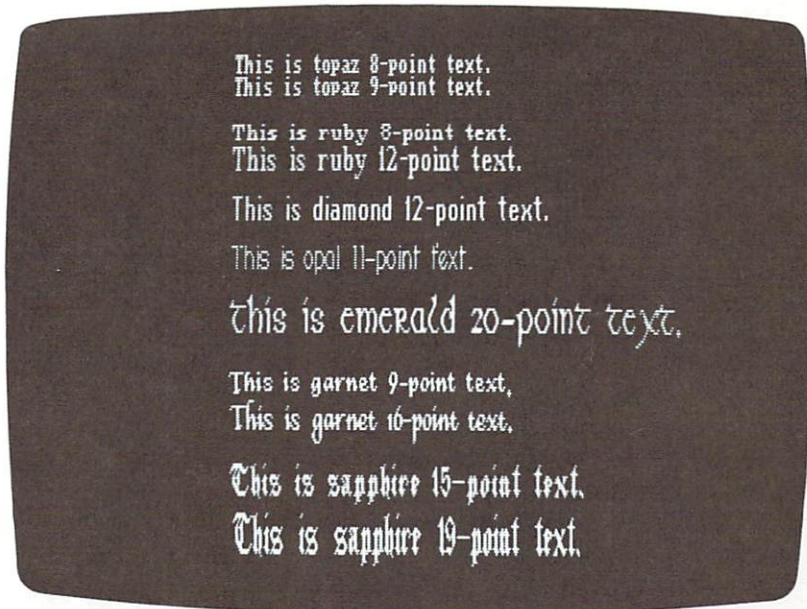


Figure 2-16.

Different character fonts on the Amiga.

Different fonts are stored on disk until the Amiga needs them, when they're transferred to RAM to be used to create the text on display. As software developers design new fonts, you'll be able to buy them on a floppy disk, load them into the Amiga, and call them in by name through the program you're using.

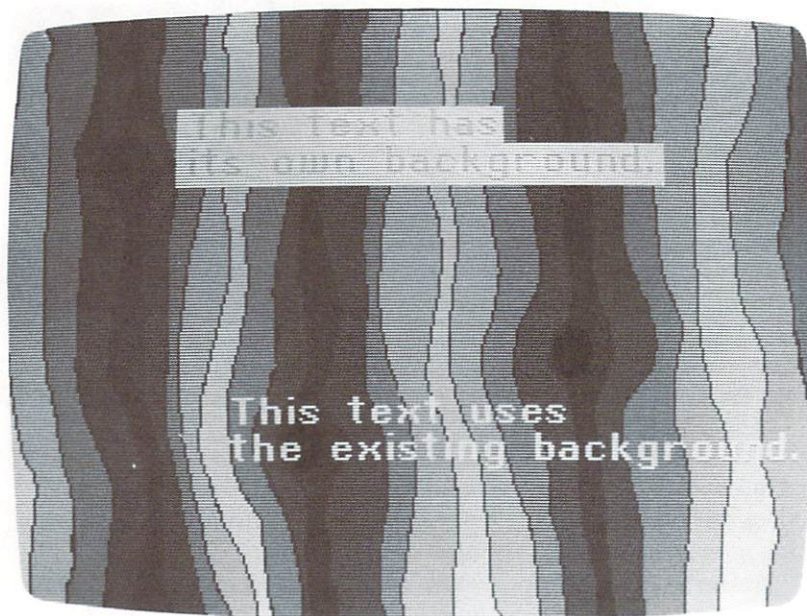
Advanced Amiga programmers programming in C or assembly language can create their own fonts by drawing the pattern for each character and storing the data on disk. This is very useful for creating characters for Russian, Greek, Hebrew, or other languages that use different alphabets. This ability is also handy for creating special mathematical or logical symbols. Watch for future commercially available software that lets non-programmers create their own fonts, too.

Once a font is loaded in memory, the Amiga can use its system software to alter the characters for emphasis. It can, for example, stretch them out twice as wide, italicize them, underline them, make them thicker, or invert their colors. It can also color the characters any one of its possible 4096 colors. You can use some of these effects in word processors like Textcraft and in graphics programs with text like Deluxe Paint.

The Amiga also controls the background of the characters. It can use the existing picture on the screen as background, or it can create a contrasting background. Characters with a contrasting background look like strips of letters pasted on a telegram. Characters using the existing background blend in with their surroundings. In Figure 2-17, you can see the characters with both types of backgrounds.

Figure 2-17.

Characters at the top of this display are put over the underlying picture with their own contrasting background. Characters at the bottom of the display use the underlying picture for their background.



SCROLLING PICTURES

The Amiga has several other graphics features currently not available to anyone but advanced programmers using C or assembly language. These features let the Amiga store a picture in memory that is too high and wide to display completely on the monitor screen at one time. In such a case, the Amiga displays just one section of the picture at a time, using the monitor screen as a window on the picture. In Figure 2-18, you can see how this feature works.

Although you can see only one section of the picture at a time on the monitor, the Amiga can still bring any section of the picture into view and so display the entire image in pieces. It can jump quickly from section to section, or it can smoothly scroll the picture into view in small increments. This makes the picture look like it's sliding by under the monitor. The Amiga can scroll a picture at any speed, fast or slow. It's not limited to up and down or left and right, but can also scroll diagonally.

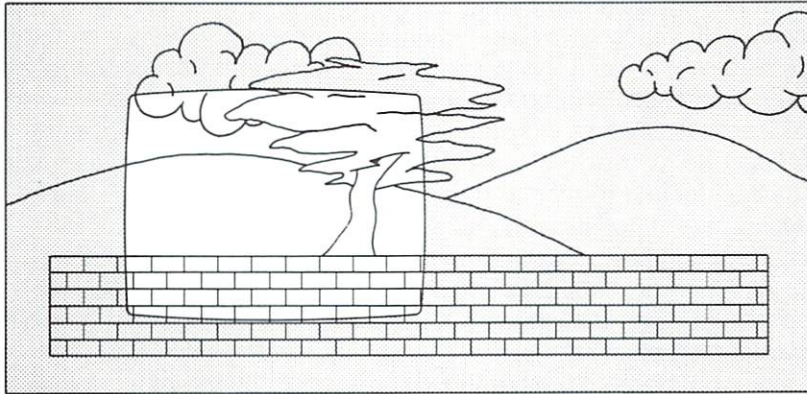


Figure 2-18.

The Amiga's monitor serves as a window on a section of a large picture stored in the Amiga's memory.

A good example of useful scrolling is writing a C program that stores a map as a large picture in the Amiga's memory. The monitor shows just a section of the map at one time, but you can instruct the program to scroll the map north or south, east or west, to see sections that are off the edge of the screen.

MIXING TWO PICTURES

The Amiga has another useful advanced feature: It can store two pictures at once in its memory and place one picture on top of another. When the Amiga overlaps two pictures, it makes one the foreground picture and the other the background picture. Normally, the foreground picture completely covers the background picture, but the Amiga can use a "transparent" color on some of the pixels in the foreground picture. Any transparent pixel in the foreground picture lets the background picture show through. Figure 2-19 shows how this works.

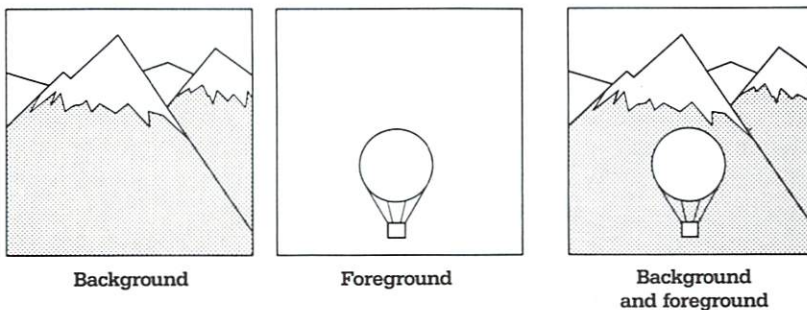


Figure 2-19.

The background and foreground pictures are stored as separate images in the Amiga's memory. When these images are displayed, the foreground picture is superimposed over the background picture.

Superimposing one picture on top of another makes it easier to change the images you see on the monitor screen. The background picture can be used like the backdrop in a theater: It's drawn only once, and doesn't change on the screen. Any elements of the monitor display that will change or that move can be drawn in the foreground picture. Since the Amiga doesn't have to keep changing the background display as the foreground changes, it has a much easier task of updating the monitor display, since it only has to re-create the foreground.

Although overlapping pictures is a feature usually available to advanced programmers using C or assembly language, you can use overlapping pictures whenever you make a video using the animation program Deluxe Video (featured in Chapter 11).

PRINTING AN AMIGA PICTURE

As you read earlier, turning a picture on the monitor screen into a picture printed on paper is no easy task. It requires color and resolution translation that is different for each printer on the market. The software that takes care of the translation is called a printer driver.

The Preferences tool on the Workbench offers you a choice of about a dozen printer drivers that the system software can use to match different printers you might connect through the Amiga's serial or parallel port. You can choose one to match whichever printer you have connected to your Amiga. As new printers come out, the manufacturers and Commodore can write new printer drivers and make them available to you. You can then load the new drivers through Preferences so that the Amiga can use the new printers.

Having a printer driver active in the system software versus having to have one built into each application is very convenient. Some computer systems force software writers to write their own printer drivers if they want their application to use a printer. This forces users to be cautious when they buy new programs that use a printer—will the program drive the printer they own? If you have the proper printer driver set for your printer in Preferences, the Amiga's own software takes care of driving the printer, so software you buy will work automatically with your printer.

Now you know the basic techniques the Amiga uses to create and display computer graphics. In the next four chapters, you'll learn how to create pictures using an application program, Deluxe Paint, and discover how to add graphics to your own Amiga BASIC programs.



**CHAPTER THREE
AMIGA GRAPHICS
TOOLS**

Creating images with a graphics application program is one of the most enjoyable activities you can pursue with an Amiga. Graphics applications are now available for users of all skill levels. They allow you to produce colorful pictures with ease and speed, and give you the power to create images of surprising complexity and subtlety. This chapter features a graphics application called Deluxe Paint, and introduces you to two more: Graphicraft and Aegis Images.

Deluxe Paint, developed and sold by Electronic Arts, is one of the most versatile graphics applications programs available. In this chapter, you'll see how to use the advanced features of Deluxe Paint. Since Deluxe Paint includes a manual that describes its individual features, this chapter doesn't duplicate what you can read there. Instead, it shows you how to combine those features to achieve practical results. It also gives you some hints that will make working with Deluxe Paint as easy as possible.

Later sections in the chapter show you how to print and photograph your Deluxe Paint images. At the end of the chapter, you can read about two of the other graphics programs created for the Amiga, and about hardware that can make your Amiga images look clear and colorful.

MASTERING DELUXE PAINT

Deluxe Paint shares many features with other graphics programs: You can choose different brush shapes from a menu of brushes; you can draw circles, squares, lines, and ovals with special tools; you can use a grid to keep your lines straight and the objects in your pictures aligned; you can magnify sections of your picture for detail work; and you can choose from a variety of fonts to add text to your picture.

Deluxe Paint also has some very powerful features that take it beyond other graphics programs. It can create pictures in three different resolutions—320-by-200 pixels, 640-by-200 pixels, and 640-by-400 pixels—and it allows you to design and alter your own multicolored brushes. Anything you draw on the Deluxe Paint screen can be selected with the brush-selection tool and used as a custom brush. Once you have selected a custom brush, you can create boxes, draw freehand lines, or do anything else with it that you can do with simpler standard brushes. You can also alter the brush by changing its size, shape, and color, or by flipping, rotating, or bending it.

Once you have the brush you want, you can use it in conjunction with different brush modes to add texture or to affect the colors on the screen in various subtle and unsubtle ways: shading, blending, and smearing colors, painting with a single color, and cycling through all the colors on your palette. Using custom brushes effectively is very important if you want to get the most out of Deluxe Paint.

Before you do anything with Deluxe Paint, the first thing you should do is to create a work disk where you can store your Deluxe Paint pictures.

CREATING A WORK DISK

You don't want to use your Deluxe Paint master disk to store your own paintings. For one reason, it's already almost full. For another reason, you should have the write-protect tab on the disk set to the write-protect position so you don't accidentally erase something important on the master disk and ruin your copy of Deluxe Paint. Creating a work disk will give you lots of room for your own pictures and keep your Deluxe Paint disk safe. It's simple to do. Just follow these steps:

1. Turn on your Amiga and load Workbench with the Workbench disk.
2. If you have one disk drive, remove the Workbench disk and insert a blank disk (or one you don't mind having erased) in the disk drive. If you have two drives, insert the blank disk in the external drive.
3. When the icon for the blank disk appears on the Workbench screen, select it and choose the **Initialize** command from the **Disk** menu, then follow the directions that appear on the screen to initialize it.
4. When the disk is initialized, double-click the second disk icon to open it up and look at its contents. You should see a trashcan there.
5. Open the Workbench disk icon to see its contents. When it's opened, drag the drawer icon labeled **Empty** from the Workbench disk window to the window for your new disk, to make a copy of the empty drawer on your disk.
6. Once the empty drawer is copied to your new disk, close the Workbench window, then select the empty drawer icon on your disk window and choose the **Duplicate** command

from the **Workbench** menu to duplicate the empty drawer. Do this twice more to make a total of three copies of the empty drawer.

7. You should now have four drawer icons in your disk window. The new icons may be stacked one on top of the other, so you may have to drag them around to see them all. To name each drawer, select the icon for the drawer, then choose the **Rename** command from the **Workbench** menu. When the title strip appears in the middle of the screen, click in the strip to select it, press the DEL key several times to erase its contents, then type in a new name and press RETURN when you're finished. (Use the cursor and BACKSPACE keys to correct mistakes if you need to.) Name one of the drawers **lo-res**, another **med-res**, another **hi-res**, and the last **brush**.
8. Select the icon for your disk, choose the **Rename** command, and follow the instructions in step 7 to give your disk a descriptive name.

You now have a Deluxe Paint work disk. When you save pictures and brushes on this disk from Deluxe Paint, Deluxe Paint will automatically use the four drawers you just created on the disk. It stores all your low-resolution pictures in the **lo-res** drawer, your medium-resolution pictures in the **med-res** drawer, and your high-resolution pictures in the **hi-res** drawer. It stores your custom brushes in the **brush** drawer.

To save you the trouble of having to create another work disk this way, you can set this one aside as a master empty work disk. Whenever you want to create a new empty work disk, just copy your entire work disk master using the directions in the *Introduction to Amiga* manual.

CHOOSING SCREEN RESOLUTION AND DEPTH

When you boot up your Amiga with Kickstart and then insert your Deluxe Paint disk, you type the command **dpaint** at the 1> prompt and press RETURN. The Amiga then loads Deluxe Paint. It comes up with a 320-by-200 resolution screen with 32 colors. If you want to create pictures using different screen resolutions, you can load Deluxe Paint with a different command. Typing **dpaint med** loads a version of Deluxe Paint that uses a 640-by-200 resolution screen; typing **dpaint hi** loads a version that uses a 640-by-400 resolution screen. In both the high- and medium-resolution versions of Deluxe Paint, you get a maximum of 16 colors.

You can also choose the number of colors available to you in Deluxe Paint when you load the program by choosing the number of bit planes you want to use for your painting. You'll learn more about bit planes in Chapter 4. For now, all you need to know is that the number of bit planes you use determines the number of colors available to you, and that the more bit planes you use, the more RAM you use. Five bit planes give you 32 colors, four bit planes give you 16 colors, three bit planes give you 8 colors, two bit planes give you 4 colors, and one bit plane gives you 2 colors. To assign bit planes, just type the number of bit planes you want after the **dpaint** load command. For example, to get a version of Deluxe Paint with a high-resolution screen three bit planes deep (8 colors), you'd type the command **dpaint hi 3**. For a low-resolution screen two bit planes deep (4 colors), you'd type **dpaint lo 2**.

Why would you want to be able to use different screen resolutions and bit-plane depths? The most important reason is that you can save the images you create in Deluxe Paint on disk and then use them with other programs. Some of those other programs require the images to use a specific resolution and bit-plane depth. For example, Deluxe Video (an animation program discussed in Chapter 11) uses low-resolution images that are three bit planes deep.

Another reason for choosing between low- and high-resolution screens is that there are distinct advantages to both resolutions. If you paint low-resolution pictures, you get to work with 32 colors. A wide range of colors in a low-resolution picture can sometimes make a picture look much more detailed and realistic than a high-resolution picture using fewer colors. If you paint high-resolution pictures, the curves and diagonals in your picture will look much smoother, and you can create finer textures with the thin lines and greater resolution available. Although the screen flickers slightly in high-resolution mode (you'll learn why in the next chapter), if you print or photograph your picture, the flicker is of no consequence.

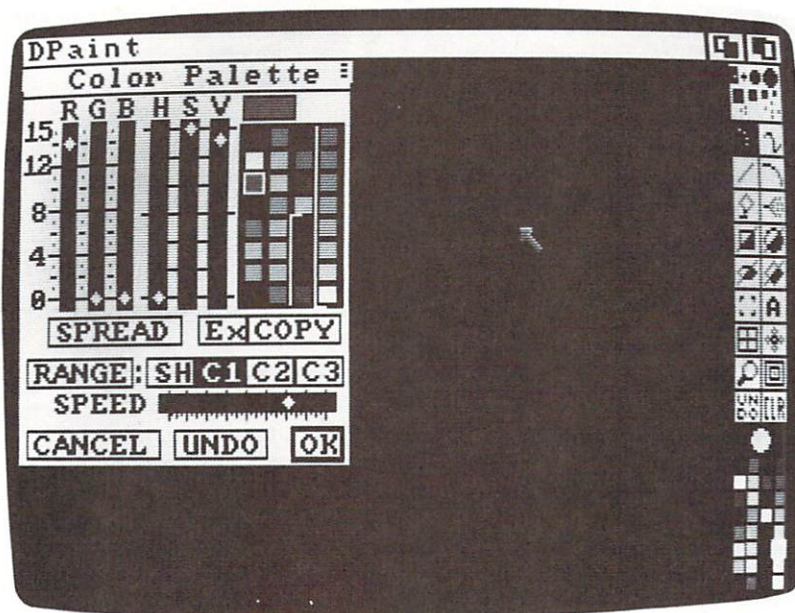
DESIGNING YOUR OWN PALETTE

After you've loaded Deluxe Paint and before you start painting, you should take some time to decide on the colors you want to use. Deluxe Paint has its own set of default colors you can use, but you might want a different color palette for your particular pictures. For example, if you're painting a forest scene, a wide range of greens and browns would be more useful than some of the pinks and purples in the default palette.

To change the colors in the palette, open the palette window by pressing **p** on the keyboard. The palette window, shown in Figure 3-1, appears. You can alter any individual color on the palette by selecting it with the pointer, and then setting the sliders to the left of the colors. There are two sets of three sliders—you can use either set to change a color. The first set, labeled **RGB**, sets the red, green, and blue components of the selected color; the second set, labeled **HSV**, sets the hue, saturation, and value of the selected color. Value is Deluxe Paint's term for intensity—the value slider sets the intensity of the selected color.

Figure 3-1.

The palette window.



When you change a color using either set of sliders, Deluxe Paint automatically adjusts the other set of sliders to match the changes you've made in the color. Which set you use is up to you: **RGB** is easier to use if you like to think in terms of mixing primary colors; **HSV** is easier to use if you like to choose a hue with the hue slider, and then lighten and darken it with the saturation and value sliders. If you aren't familiar with RGB and HSV (frequently called HIS) color creation, be sure to read Chapter 2.

The five boxes labeled **RANGE**, **SH**, **C1**, **C2**, and **C3** control the four ranges of colors Deluxe Paint uses when you paint using special brush modes (more on that later in this chapter). The four different ranges of colors are the shade range (**SH**), color-cycle range 1 (**C1**), color-cycle range 2 (**C2**), and color-cycle range 3 (**C3**). Deluxe Paint works with the **SH** range when you use the **Blend** and **Shade** brush modes; it cycles through the colors in the **C1**, **C2**, or **C3** ranges when you use the **Cycle** brush mode or the **Cycle** command in the **Picture** menu. All these ranges have been preset—you can view any of them by selecting **SH**, **C1**, **C2**, or **C3**. A white bracket appears on the palette to show you what colors are included in the range and where it begins and ends.

You can change the default range settings to suit your needs. To change a range of colors, first select which of the four ranges you want to reset. Next, select any of the colors in the palette as the starting color in the range, then select **RANGE**. A **TO** pointer appears on the screen. Use it to select any color in the palette as the last color in the range. The **TO** pointer will then disappear, and the new range is set.

The shade range works best when you set it to coincide with a spread of colors that ranges from dark to light. A good example of this is the range of grays that are the last 12 colors in Deluxe Paint's default palette. When you use the **Blend** and **Shade** brush modes, Deluxe Paint can move up and down in this range to darken or lighten the colors you have on the screen.

When you ask Deluxe Paint to cycle colors, it cycles through the colors in the selected cycle range, using the speed set for that range in the **SPEED** slider located just below the range controls. You can change the speed of a color cycle by first selecting it and then setting the **SPEED** slider.

Since there are three different cycle ranges, you can set one range inside another range for some very bizarre effects when you cycle the colors on the screen. If you want to get rid of any range in the palette, just select the range, select a color, select **RANGE**, then use the **TO** pointer to select the same color again—effectively creating a one-color range.

When you've set the palette to your taste, you can select the **OK** command to apply it, or, if you have second thoughts, you can select **CANCEL** to go back to the palette you were using before you started changing colors or ranges. No matter what changes you've made, don't worry about losing Deluxe Paint's default palette. You can always get it back when you need to by choosing **Default Palette** from the **Picture** menu.

CREATING PRECISE DRAWINGS

As you use the Deluxe Paint drawing tools, you may find yourself trying to place line ends, rectangle corners, circle edges, and other object boundaries in precise locations so they match up with other objects on the screen. You can always use the magnifier tool to get a closer look at what you're doing, or you can scrunch up close to the screen and squint at the individual pixels, but there are easier ways to click your figures into place.

Positioning the brush with the Coordinates command

One of the easiest ways to position your brush on the screen is to choose the **Coordinates** command from the **Prefs** menu. When you do, you'll see numbers appear in the right side of the title bar that give you the location of the pointer that moves your brush. The numbers measure the distance in pixels from the upper left corner of the screen to your brush pointer—the left number is the number of pixels over and the right number is the number of pixels down. If you note the coordinates when you place the end of a line, the center of a circle, or the corner of a rectangle on the screen, you can use the coordinates to start the next figure at the same spot. For example, say you want to create a series of lines radiating from a central point. When you draw the first line using the straight-line tool, you note the coordinates of the beginning of the line, and use that as your central point. By beginning the other lines at the same coordinates, you can be sure all your lines will radiate from the same point.

The coordinates work differently whenever you hold a mouse button down and drag the cursor across the screen. Instead of giving the position of the cursor in distance from the upper left corner of the screen, it gives the distance of the cursor from the spot where you first pressed the mouse button. If you're creating an object with a tool, this lets you measure just how large the object is. For example, when you create a rectangle, if you want to make it 10 pixels wide and 16 pixels high, you can read the coordinates as you set the opposite corner of the rectangle by moving the pointer until you get the measurements you want.

Drawing straight lines with the SHIFT key

Another aid to cursor accuracy is the SHIFT key. If you hold down the SHIFT key while you roll the mouse, Deluxe Paint will move the brush in a horizontal or vertical direction only, depending on whether you move the mouse horizontally or vertically while holding down the SHIFT key. If you want to move the brush in the other direction, you have to release the SHIFT key.

This method is very useful for drawing straight lines or for moving a brush in a straight line across the screen. For example, say you want to draw a straight line from one side of the screen to the other without moving up or down. Select the line-drawing tool, then move the mouse pointer to the place where you want the line to begin. Before you press the left mouse button to begin drawing the line, hold down the SHIFT key. Then hold down the left mouse button and start rolling the mouse to the side. As you roll, the SHIFT key keeps the line on a strict horizontal path, even if your hand moves the mouse up and down as you roll it.

Aligning images with the grid

The grid is even more useful for quickly matching up line ends and objects. When you select the grid tool on the control panel, it sets up a grid of invisible lines on the screen, and restricts the location of the pointer to the intersection of those lines while you use many of the other tools on the control panel. The grid works with the dotted freehand-drawing tool by restricting each image of the brush it puts down to the intersections of the grid. It works with the straight-line and curve tools by restricting both ends of the line each creates—the lines must begin and end at grid intersections. The grid similarly restricts to intersections the corners of a rectangle that is created with the rectangle tool, and the center and the borders of shapes that are created with the circle and oval tools.

If you turn on the grid and begin to draw with these tools, you can very easily line up the figures they create. For example, if you want to create a box with a dome on the top, you can turn on the grid and create the box with the rectangle tool. The grid will limit its corners to the grid intersections, so the cursor will jump from intersection to intersection as you move it around the screen. After you create the box, you can select the curve tool. Since the grid limits cursor movement to the grid intersections, it's easy to position the pointer on an upper corner of the box—simply position the center of the crosshairs over one corner. Hold down the left mouse button and drag across the top of the rectangle to the other upper corner, then release the mouse button. The ends of the curve will be directly on top of the corners. You can then move the mouse to bend the curve as you want it.

If the default grid intersections (marking 8-by-8 pixel squares) are too close together or too far away for your purposes, you can change the size of the grid by selecting the grid tool with the right mouse button. A small section of the grid, called the sizing grid, appears on the screen. The coordinates will appear in the right side of the menu bar and show you the location of the upper-left corner of the sizing grid on the screen. By holding down the left button

and dragging the mouse, you can change the size of the grid. As you move the mouse, the coordinates will change to show you how many pixels wide and high each square of the grid is. Once you've dragged the grid to the size you want, you can release the left button. The grid will disappear, and the next time you turn on the grid tool, the new size will determine where the grid intersections are located.

By default, the grid is aligned with the top and left boundaries of the screen, but you can reposition the grid intersections if you want to. First, select the grid tool with the right mouse button again. The sizing grid will appear on the screen again, but this time you will be positioning it on the screen to determine where the actual grid lines themselves will be—not the size of the individual squares. Place the intersections of the sizing grid where you want the grid lines to be positioned, then click the left mouse button, and the entire grid is realigned to match the position of the sizing grid on the screen.

When you use the grid with the text tool, it restricts where you can place the text cursor with the pointer. Once you start typing, the letters aren't restricted by the grid, but if you move the text cursor by clicking elsewhere on the screen, its new position is restrained by the grid. By stretching the grid to the size you want, you can use the grid intersections as tab stops and line spacing. Just type what you want, then relocate the text cursor with the pointer. The grid makes it easy to line up columns and lines of text.

One of the most useful applications of the grid is in creating background patterns. You can try it out by first loading the picture **Patterns** from the Deluxe Paint disk. In the upper right corner of the picture are eight patterns in yellow boxes. You can select any one of those patterns as a custom brush, and then duplicate it all over the screen using the grid to line up the brushes.

Try it out. Use the brush-selection tool to select the brick pattern by selecting all of the pattern inside the yellow box without including any of the yellow border. If you watch the coordinates as you set the brush, you'll see that the box is exactly 19 pixels wide by 11 pixels high. Press **j** on the keyboard to switch to the other drawing screen, where you can paint in a blank screen without covering the other patterns. Now resize the grid so it's one pixel wider and higher than your brush: Set it to 20 pixels wide by 12 pixels high. Once the grid size is set, turn on the grid, make sure the brush is set for dotted freehand drawing, and start painting. Perfectly aligned blocks of the brick pattern will then start to fill the screen.

Once you fill the entire screen with bricks, try painting other things on top of the bricks. If you paint something you don't like, and if clicking **UNDO** doesn't erase all of it, you won't be able to erase it using the background color, because the brick background isn't one single color. Instead, turn on the grid if it isn't already turned on, then grab any section of brick as a brush and start painting over what you draw. If you haven't changed the grid since you laid down the pattern, the brush will erase your figure by filling it in with new bricks aligned with the old bricks.

THE BRUSH MODES

A lot of the fun in using Deluxe Paint comes from creating custom brushes by cutting them from a picture already on the screen, then using the commands in the **Brush** menu to flip, rotate, reshape, and recolor the brush until you get exactly the shape and color you want. Once you have a custom brush, or even a simple brush selected from the control panel, the brush mode you choose from the **Mode** menu determines the way that brush draws on the screen. The **Object**, **Color**, and **Replace** modes give you some options for painting with a custom brush. If you want to add texture to your painting, you can use three other brush modes that affect the colors already on the screen: **Smear**, **Shade**, and **Blend**. Another brush mode, **Cycle**, can be used to create special effects. The following sections describe each of the different brush modes in detail. You'll get to try out all the brush modes in a later example section in this chapter.

The Object mode

Whenever you use the brush-selection tool to create a rectangle on the screen and copy the contents as a brush, Deluxe Paint automatically uses the **Object** brush mode. In the **Object** mode, Deluxe Paint looks to see what the current background color is. Any pixels in the rectangle that are colored with the background color become transparent in the brush. In other words, the custom brush rectangle doesn't pick up any background pixels. When you paint with the brush in **Object** mode, you paint with all the different colors in the brush, using the shape of the object you picked up without background colors.

The Color mode

Once you've created a custom brush—for example, if you draw a rainbow on a screen with a white background, then use the brush-selection tool to pick it up as a custom brush—Deluxe Paint automatically uses the arc shape of the rainbow with all its colors as a brush in the **Object** mode. Then, if you choose the **Color** brush mode, Deluxe Paint will turn the brush you picked up into a

silhouette, coloring it with a single color: the current foreground color. If, in our example, the foreground color is set to green, Deluxe Paint will use the arc shape of the rainbow, but will fill it with only one color: green.

The Replace mode

After you've selected a custom brush, another option you have is the **Replace** brush mode. In this mode, Deluxe Paint will include all the background color in the brush that was treated as a transparent color before. In the custom brush of the last example, the rainbow would be there with all its colors, but so would be the background white that was picked up in the brush-selection rectangle. When you paint with a custom brush in the **Replace** mode, you paint with a full rectangle; the background color is used as well as the other colors in the brush.

The Smear mode

The **Smear** mode is the Deluxe Paint equivalent of using your finger to smear different colors of oil paint together. When you paint in **Smear** mode, the brush drags around the pixels already on the screen, mixing them together. You can use a standard or a custom brush—if you use a custom brush, Deluxe paint uses the same silhouette shape of brush it would use in **Color** mode.

The Shade mode

In the **Shade** mode, the brush affects only pixels whose colors are included in the shade range on the color palette. All colors outside the shade range aren't affected by the brush. You can use either a standard or a custom brush. The custom brush uses only the silhouette shape of the object in the brush.

If you use the **Shade** mode, whenever you hold down the left mouse button and pass the brush over pixels colored with shade-range colors, the pixels under the brush change to the next color down in the shade range. If you hold down the right mouse button, the pixels change to the next color up in the shade range.

If you created a shade range of colors in the palette window that progresses from dark to light, you can use the left button to lighten the pixels under the brush and the right button to darken them. The pixels won't change to any color beyond the ends of the shade range, so you can't lighten or darken them any further than the lightest and darkest colors in the shade range.

The Blend mode

The **Blend** mode works something like the **Smear** mode: When you drag the brush in **Blend** mode over two different colors that are both in the shade range, the brush drags some of the first color into the second color. The difference is that in **Blend** mode, the brush doesn't just drag pixels of one color into an area of another color: The brush compares the colors it's dragging to the colors it's passing over, and lays down a color halfway between the two colors in the shade range. (If the two colors are adjacent in the shade range, Deluxe Paint will use the color you started blending from.) When you work with a shade range that progresses from dark to light, this means that you can use the **Blend** mode to blur the borders between two different colors, using all the intermediate shades of color between them to make a smooth transition from one to the other. This sounds much more complicated than it really is. If you try the example later in the chapter, you'll see just how it works.

The Cycle mode

The **Cycle** mode is great for splashy special effects. When you paint in the **Cycle** mode, your brush cycles through all the colors you set in one of the cycle ranges, and leaves a trail of different colors on the screen. To choose any one of the three cycle ranges (**C1**, **C2**, or **C3**), choose any one of the colors in the control panel that belongs to the cycle range you want, then start painting. You can also set the speed of the color cycle using the **SPEED** slider for that color range in the palette window.

If you use the **Cycle** mode to create boxes, circles, lines, and other shapes using the tools on the Deluxe Paint control panel, each object you create is just one color, but each new object is a new color in the cycle range.

Once you've painted using cycle-range colors, you can make all the colors on the screen that are in the cycle range cycle through the entire range by pressing the **TAB** key—try it!

EXAMPLES USING THE DIFFERENT BRUSH MODES

Now that you know what the different brush modes do, you can try some examples that use the brush modes for various effects. These examples are all fairly simple, but some produce sophisticated results that might give you ideas for pictures of your own. These examples use the lo-res Deluxe Paint mode, so before you try the examples, you should start the program by typing **dpaint** at the **1>** prompt.

Using the Object and Color modes to outline a figure

When you create an object using colors that don't contrast well with the background color on the screen, you might want to outline your object with another color to make it stand out. For example, an orange object on a white background will stand out better if you outline the object in black. You can always outline your object by carefully drawing around the edges, but that takes a lot of time if the object is at all complicated. By selecting the object as a brush and using the **Object** and **Color** brush modes, you can create an outline for that object in seconds, no matter how complex its shape. Follow these instructions to try it out:

1. Use the right mouse button to select white as the background color, then select **CLR** in the control panel to clear the screen and fill it with white.
2. Choose **Load Fonts** from the **Font** menu, then choose a large font (**sapphire-19** works well for this example).
3. Select the text tool, put the text cursor on the screen, and click to position the text cursor.
4. Choose light blue from the palette, then type a message on the screen. The message should appear in large light-blue characters.
5. When you're finished with your message, select the entire set of characters as a custom brush.
6. Choose **Color** from the **Mode** menu, then choose black from the palette to turn your brush black.
7. Choose **Coordinates** from the **Prefs** menu so you can see the coordinates in the title bar.
8. Select the hollow-rectangle tool (upper left corner of the rectangle tool).
9. Move the brush to a blank area of the screen, then hold down the left button and drag down and right until the coordinates measure 2 by 2 pixels. The result will be a somewhat illegible set of black blobs on the screen, similar to what you see in the right half of Figure 3-2.

10. Choose **Object** from the **Mode** menu to turn your brush back to its original light blue color.
11. Position your brush in the middle of the "rectangle" you just created so you can see the text with a black outline. Click the left mouse button to copy the brush there. When you move the brush away, voila! You have light-blue text outlined in black, something like what you see in the lower part of Figure 3-2.

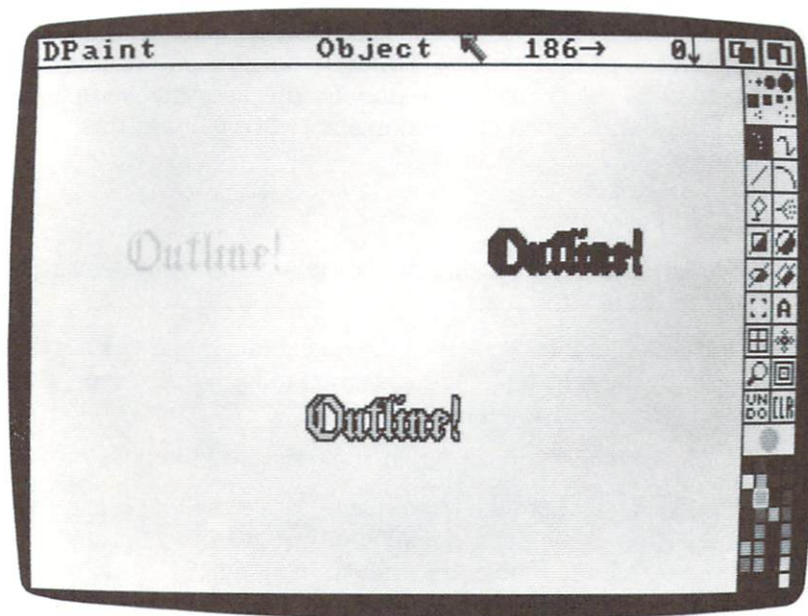


Figure 3-2.

The "rectangle" to the right of the screen was created by a text brush. The lower part of the screen shows the original text brush positioned in the middle of the rectangle to create outlined text.

12. Select the outlined text as a new custom brush, then save it to disk. You can use it in a later example.

In this example, when you used your custom brush in the **Object** mode, Deluxe Paint always used the original colors (light blue in this case) that the brush had when you first selected your custom brush. Once you change your custom brush to **Color** mode, you can choose any color you want. (In this example, in step 6 you could choose any color you wanted to outline your message.) As soon as you choose the **Object** mode again, your brush will return to its original colors.

Creating a sandy background with the Smear mode

Most of the tools in the control panel create objects of a single color. If you want to create patterns that mix a lot of different colors in a small area, you can spend a lot of time trying to do it with standard tools. For example, consider trying to draw a picture of beach balls on a sandy beach. Drawing each grain of sand on the beach—alternating light, dark, and in-between—can be a very tedious job. You can use a brush in the **Smear** mode to perform the job with much more speed and ease.

In this example, make sure that you're using the default palette with the default setting for the shade range, so it includes the last 12 colors on the palette, ranging from dark gray to light gray. While these particular colors aren't necessary for this specific example, you can use these results in a later example that will use this shade range.

1. Clear the screen.
2. Choose a circular brush with the right mouse button and size it to 10 by 10 pixels.
3. Pick each of the 12 colors in the shade range and put a dab of it on the screen so the dabs touch to form a circle, as you can see in the left side of Figure 3-3.
4. Choose a small circular brush.
5. Choose **Smear** from the **Mode** menu, then move the brush to the center of the dabs, hold down the left mouse button, and start smearing. Smear in a circular motion from inside the cluster of dabs so you get a little bit of each color smeared in the center. The result should look like the right side of Figure 3-3.
6. Once you have a well-mixed smear in the center, start smearing from the center toward the edges. This evens out the texture and makes more of it as you push pixels away from the center.
7. When you have a fairly large section of sand texture, select the center of the section as a custom brush, then paint over the entire screen to fill it with the sand texture. The result should look like Figure 3-4.
8. Save your screen to your data disk as a picture by choosing the **Save As...** command from the **Picture** menu. You'll use the sand background for a later example.

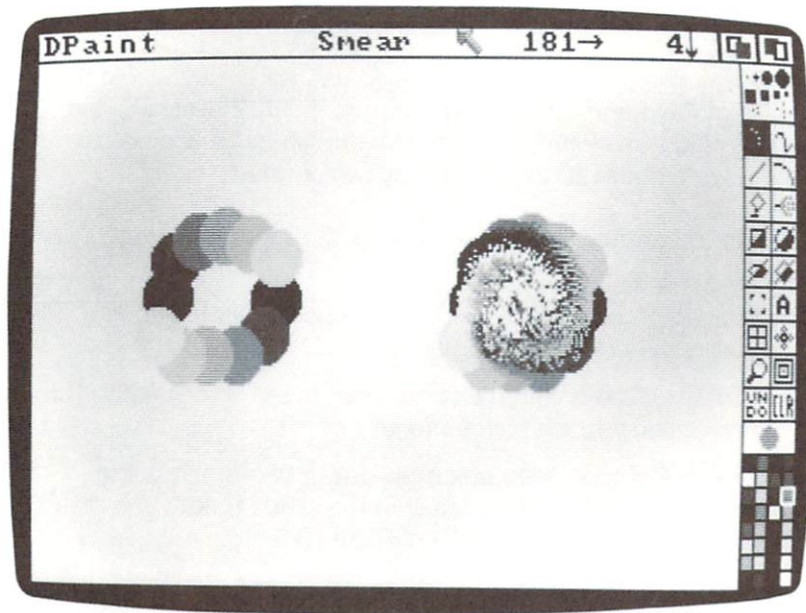


Figure 3-3.

The colors on the left side of the screen are dabs of color about to be smeared. The right side of the screen shows the colors after smearing.

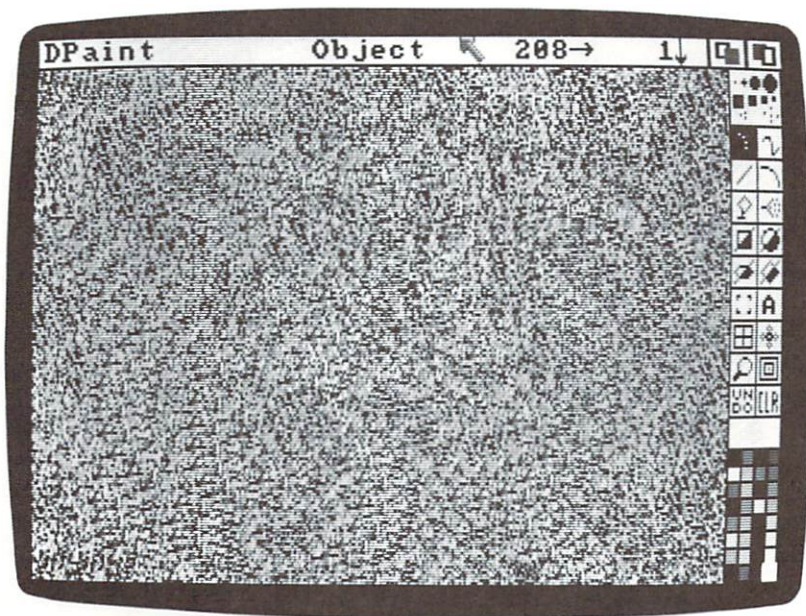


Figure 3-4.

A screen full of sand texture.

Creating rainclouds with the Blend mode

If you've ever looked at rainclouds, you'll notice that they have a wide range of grays—from light gray to very dark gray where rain is pouring down from them. If you want to create closely shaded clouds with Deluxe Paint, you can use a brush in **Blend** mode to create a wide range of grays from just two original shades.

1. Clear the screen. (The background should still be white.)
2. Open the palette window, make sure the 12 shades of gray at the end of the palette are the only 12 colors in the shade range, then close the palette.
3. Create a filled box in the upper third of the screen using the darkest gray in the shade range.
4. Create a second filled box in the lower two-thirds of the screen using the lightest gray in the shade range. The result should look like the left side of Figure 3-5.
5. Pick the largest circular brush in the control panel (the one to the far right), then choose **Blend** from the **Mode** menu.
6. Move the brush to the dark area in the upper third of the screen, hold the left mouse button down and slowly move the brush down and slightly toward the right into the light area of the screen. The dark area should smear and blend into the light area. Repeat the stroke many times, working across the border between the two shades of gray to create a rainy look.
7. To add a tornado coming from the clouds, continually streak down from one spot in the clouds. Each stroke should extend the dark clouds lower into the light area. Keep stroking down in a curve to the bottom of the screen to create a funnel cloud. The final result should look like the right half of the screen in Figure 3-5.

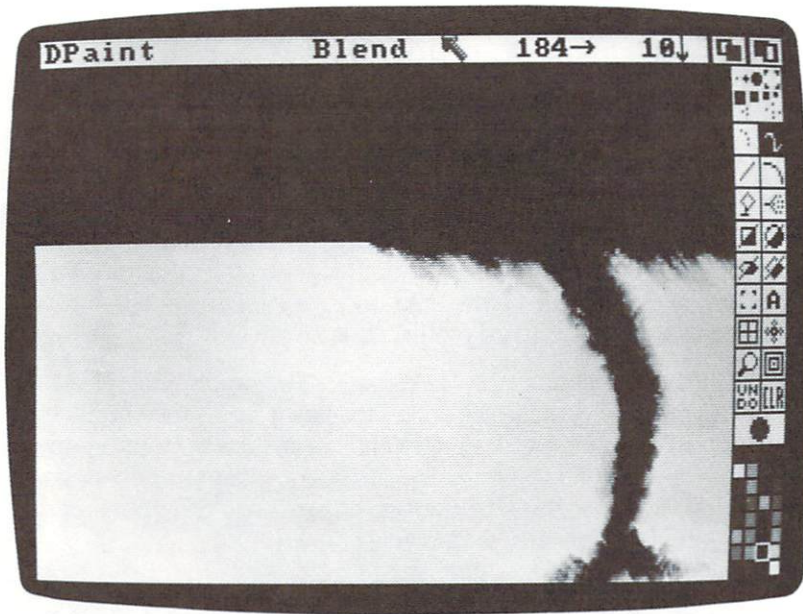


Figure 3-5.

The two gray shades shown in the left half of the screen are blended with a brush in the Blend mode to create the rain-clouds and the tornado shown in the right half of the screen.

Using the Shade mode to create a shadow

The Deluxe Paint manual explains how to create a quick drop shadow for an object: You simply pick the object up as a brush, choose **Color** from the **Mode** menu, choose black as the color, and put a copy of the black brush on the screen as a shadow for the original object. Then choose **Object** from the **Mode** menu and place the original object just above and to one side of its "shadow."

You can also use the **Shade** mode to create a more realistic shadow on a complex background if you created the background using colors graduated from light to dark in the shade range, as we did with the sandy-beach background example. This creates shadows that aren't just pitch black—they're the original background, darkened so you see a shadow over the original design of the background.

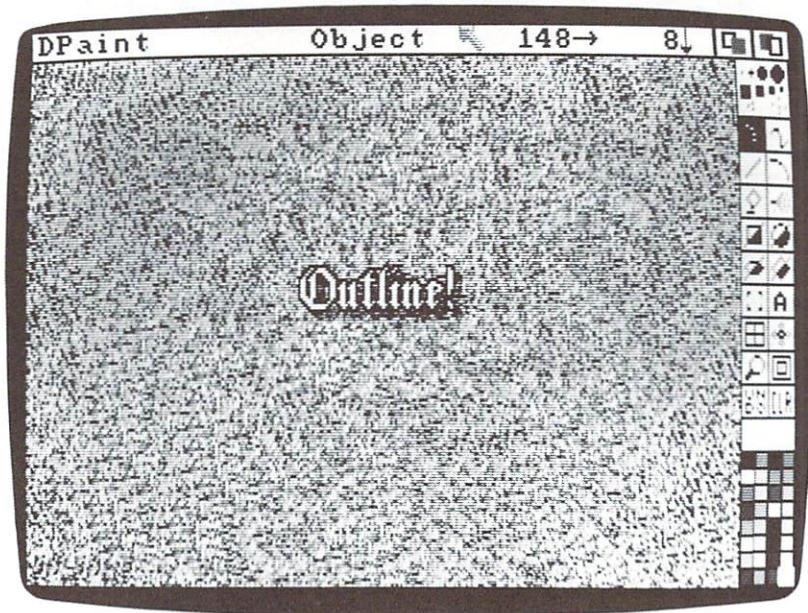
In the following example, you can create a shadow for the outlined words created in a previous example, on a sand background saved in another previous example.

1. Load the sand background you created earlier and saved as a picture.
2. Load the outlined text you created and saved as a brush.

3. Put the brush on the screen wherever you want to place the text.
4. Choose **Shade** from the **Mode** menu. The brush shows on the screen as a lighter shade of sand in the shape of your letters.
5. To darken the sand underneath the brush, click the right mouse button six times without moving the mouse. The sand gets darker with each click. When you move the brush off the darkened area, it looks even darker, since the brush in **Shade** mode always makes the area under it look one shade lighter so you can see where the brush is.
6. Choose **Object** from the **Mode** menu. The brush will appear as the original outlined letters. Place them just above and to the left of the shadow, then click the left mouse button to copy the letters there. The results should look like Figure 3-6. If you look closely at the shadow, you can still see the grains of sand, although they're darkened.

Figure 3-6.

Letters floating above a shadow in the sand created with a brush in the **Shade** mode.



You can use this shadow method on any background textured using the **Shade** mode. You can use backgrounds of woodgrain, metallic grids, fields of grass—anything that uses a range of colors with the same hue.

EASY TRICKS

As you use Deluxe Paint, you'll find many shortcuts and neat tricks to help you create pictures. To start you off right, here are three tricks that you may find useful.

Use the keyboard shortcuts

At the back of the Deluxe Paint manual is a list of keyboard shortcuts you can use in lieu of choosing tools from the control panel and commands from the menus. Open the manual to that page and lay it next to your workspace, and take the time to memorize the key shortcuts as you work. It's well worth the trouble. You'll find your work goes much faster when, for example, you can press the **x** key on the keyboard to reverse a brush on the screen, instead of opening the **Brush** menu, selecting **Flip**, and then selecting **Horiz**.

An added benefit of using keyboard shortcuts is that you can turn off the control panel and the title strip, so you can use the full screen to paint your pictures. Then, since you need only the keyboard to use the shortcuts, you won't have to continually turn the panel and strip back on to change brushes or choose new tools and colors.

Most of the keyboard shortcuts aren't difficult to remember; they're letters that usually stand for something, like **b** for the brush tool. The function keys at the top of the keyboard are another matter. To help you use them, you can make a function-key strip. Just above the function keys on the keyboard is a small indentation in the plastic; this is designed to hold a function-key strip. Just cut a piece of paper or cardboard to the right size to lay in the indentation, then write a label on the strip above each function key to remind you what the key does.

Put the monitor on its side

Since the Deluxe Paint screen is wider than it is tall, it's easy to fall into a rut and create pictures that are always wider than they are tall. Change the proportions of your workspace by turning your monitor on its side. Although it might take a little while to get used to keeping your mouse turned sideways as you roll it, and to pulling menus from the side of the screen instead of the top, you'll find that working on a screen that's taller than it is wide gives you a fresh perspective. If you print your Deluxe Paint pictures vertically on a sheet of paper (see the "Printing Deluxe Paint pictures" section later in this chapter for details), you have the added advantage of being able to see the full length of paper on the screen.

Copy pictures from magazines onto acetate

If you're not the greatest freehand sketch artist in the world, or if you are but want to save yourself some work, you can use sheets of .002 weight clear acetate (available in most art stores), a fine felt-tip marker, and some magazines to good advantage. If you lay the acetate on a magazine picture, you can trace the image on the acetate with the felt-tip marker. When you're finished, tape the acetate to your monitor screen, and follow the traced lines with the Deluxe Paint cursor to copy the image to the screen. As long as you work alone behind closed doors, no one but you will know your secret.

PRINTING DELUXE PAINT PICTURES

If you have a printer attached to your Amiga that will print graphics, you can print any picture created with Deluxe Paint by choosing the **Print** command from the **Picture** menu. Deluxe Paint then sends the data for your picture to the printer driver that was loaded with the Amiga's system software when you first inserted the Deluxe Paint disk. The printer driver translates the picture data into a stream of data that prints out your picture.

Although you don't have any control from within Deluxe Paint over the way your picture is printed, you can get quite a variety of different printouts from the same picture by changing different aspects of the printer driver from within the Preferences program.

USING THE PREFERENCES PROGRAM

To use the Preferences program, you need to type the command **preferences** at the **1>** prompt you see when you boot the Deluxe Paint disk (called the CLI screen) instead of typing **dpaint** to load Deluxe Paint. If you're already using Deluxe Paint and want to use Preferences, you must save your current picture and then quit Deluxe Paint to return to the CLI screen, where you can type **preferences** to load the Preferences program.

The Change Printer screen

The first Preferences screen that appears doesn't affect the printer driver at all. To switch to another screen where you can change the printer driver, select the box labeled **Change Printer**. The screen you see in Figure 3-7 appears on your monitor.

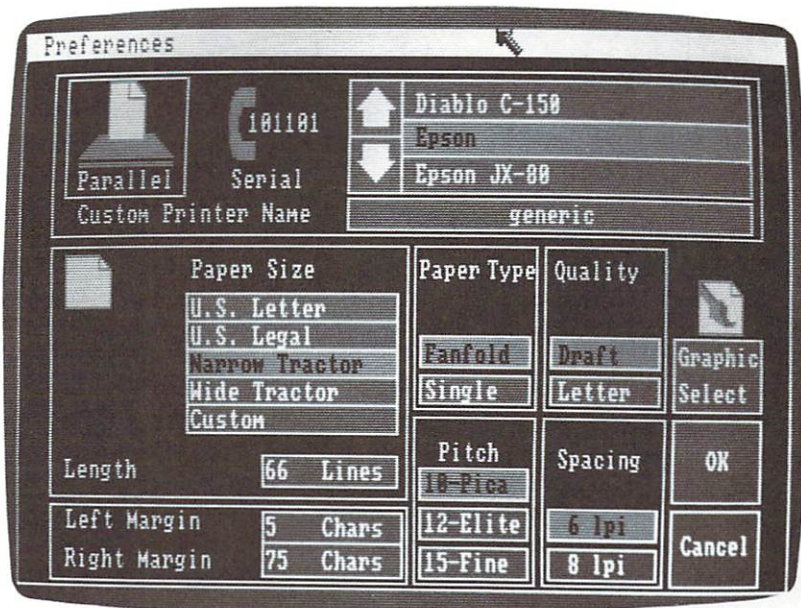


Figure 3-7.

The Change Printer screen.

The settings you see in this screen are described in the *Introduction to Amiga* manual that comes with your Amiga—you can read the manual to learn how they work. These settings are designed to aid in printing text, but some of them affect the way graphics are printed as well. The following explanations will help you use them for your picture printouts.

The **Margin** box in the lower left corner of the screen lets you set the margins of your printout. It measures the distance in characters from the left edge of the paper to the left and right borders (called margins in this box) of your printout. The spacing between the characters is set in the **Pitch** box to the right of the **Margin** box. The **Pica** setting chooses characters that measure 10 per inch, the **Elite** setting chooses 12 characters per inch, and the **Fine** setting chooses 15 characters per inch.

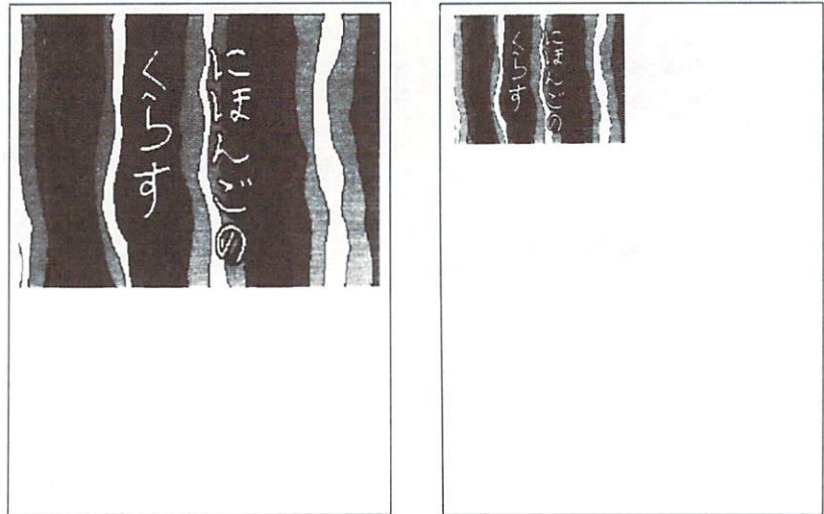
You can use these left and right border settings to determine the size of your picture. First, determine the width of your paper in characters, then determine where you want each border of the picture to appear. For example, if you use standard 8½ inch paper and you have the pitch set to 10 characters per inch, this means your paper would measure 85 characters across with no margins. If you want your picture to measure 7½ inches across so it has ½ inch of blank space on each side of it, then you would set the left

margin at 5 characters ($\frac{1}{2}$ inch from the left side of the paper) and the right margin at 80 characters (8 inches from the left side of the paper and $\frac{1}{2}$ inch from the right side of the paper).

When the printer driver sends your picture to the printer, it keeps the height in proportion to the width you set. If you print a picture using the above settings, it will look like the picture on the paper in the left half of Figure 3-8. If you reset the margins to 5 and 40, halving the width, when you print the picture the height is also halved, as shown in the right half of Figure 3-8.

Figure 3-8.

The picture on the left sheet was printed with margin settings of 5 and 80. The picture on the right sheet was printed with margin settings of 5 and 40—effectively quartering the size of the image.



The Graphic Select screen

The **Change Printer** screen lets you choose a printer driver and set the size of your paper and your printout. If you want to control the aspect and color of your printout, choose the box on the screen labeled **Graphic Select**. The screen you see in Figure 3-9 will appear.

The *Introduction to Amiga* manual describes the settings in this screen, but one setting can use a little further explanation to help you control the size of your picture printout.

In the **Aspect** box in the center left of the screen, you can choose between printing your picture horizontally or vertically. If you choose **Horizontal**, the picture will print out on the paper as you see in the left side of Figure 3-10. If you choose **Vertical**, the picture will print out as you see in the right side of Figure 3-10.

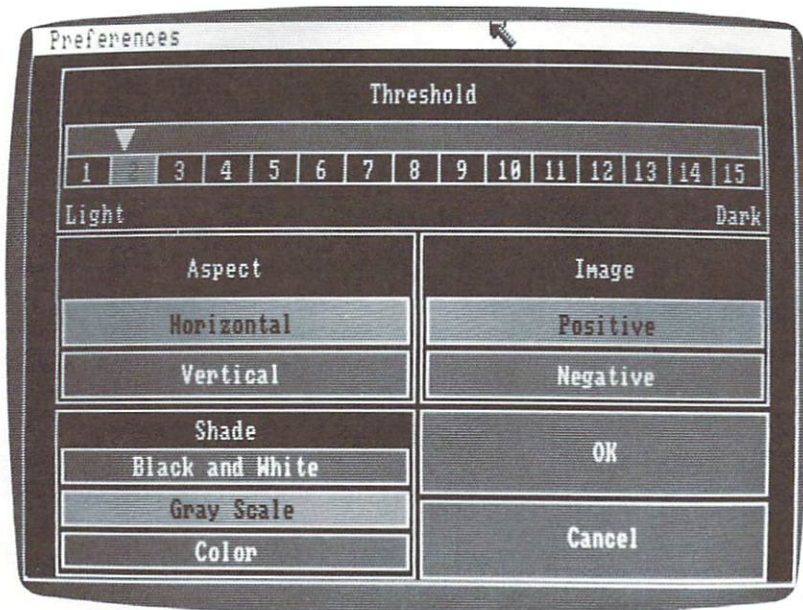


Figure 3-9.

The Graphic Select screen.



Figure 3-10.

On the left is a horizontal printout; on the right is a vertical printout of the same picture.

The two examples in Figure 3-10 use the same margin settings. Notice that the margin settings control the width of whichever side of the picture goes from left to right. This means that a vertical picture will be larger than a horizontal picture printed with the same margin settings, as you can see in Figure 3-10.

Once you've set the graphics settings as you like them, you can select **OK** to return to the **Printer Change** screen, where you can select **OK** to return to the first **Preferences** screen. Once there, if you want to use your new printer settings without changing your default printer settings, you can select **Use**, which will put them into effect without saving them to disk. If you want to save your new settings as the default printer settings, select **Save**, which will put them into effect and also save them to disk so that the next time you boot Deluxe Paint, these settings will automatically go into effect without your having to set them again using the Preferences program.

PHOTOGRAPHING AMIGA IMAGES

Photographing a painting on the Amiga monitor screen is quite simple. All you need is a camera (preferably 35mm or larger), a tripod, a cable shutter release, and a dark room (that is, a room that is dark, not a room for developing pictures). If you have a low-power telephoto lens (an 80mm lens for a 35mm camera, for example), it will help your photo, although it's not necessary.

Set your camera up on the tripod, and position it so the monitor screen fills the entire frame when you look through the viewfinder. Make sure the camera points directly at the monitor screen—if it's aiming at an angle, that will distort the picture. If you have a low-power telephoto lens, you can move further back from the screen than you could with a normal lens, and still fill up the viewfinder frame. By moving back, you reduce the amount of barrel-like distortion you get from the curvature of the monitor screen.

Once you have the camera positioned and ready, load the Deluxe Paint painting you want to photograph. Choose the one dot brush from the control panel, then choose a color used in your picture. Press the F8 key to make the crosshairs disappear, then move the dot over a pixel of the same color so it disappears from view. Press the F10 key to make the title strip and control panel disappear, and you're ready to shoot.

Before you turn out all the lights in the room, give the glass over the monitor screen a quick wipe with a cloth and some window cleaner to make sure it's clean. Turn out all the lights. Set the shutter speed of the camera to 1/15 of a second or slower, then adjust the f-stop to get the proper exposure. Use the cable release to click the shutter so you don't jiggle the camera and cause blurring at the low shutter speed you're using. Voila! You're finished. All you need to do now is have the film developed.

OTHER AVAILABLE GRAPHICS SOFTWARE

Deluxe Paint is not the only graphics application program available for the Amiga. Several other programs also make use of the Amiga's superb graphics to help you create and print out pictures. If you are interested in a graphics program for the Amiga, you should take a look at Graphicraft and Aegis Images as well as Deluxe Paint, and choose the one that fits your style of creation.

GRAPHICRAFT

Graphicraft is a graphics program marketed by Commodore-Amiga. Although you cannot create multicolored custom brushes or work on a high-resolution screen with Graphicraft, you can still use it to create excellent pictures. It just takes a little more time to do it.

One good reason for buying Graphicraft, or at least looking at it, are the sample pictures that come with it. They take full advantage of Graphicraft's ability to cycle colors to create beating hearts, storming weather maps, and flying hot dogs. If you take the time to learn how these pictures work, you can then create some very sophisticated animation using any graphics program that can cycle colors.

AEGIS IMAGES

Aegis Images is a graphics program that is sold together with Aegis Animator, a video-animation program. You can use the pictures you create in Images as objects to animate in Animator.

Although Images does not offer custom multicolored brushes and different screen resolutions, it has other special features. Where the power of Deluxe Paint is in selecting multicolored custom brushes and building pictures with them, Images concentrates on the versatile use of monochrome brushes.

You can use the Images brushes to paint with single colors or with multicolored patterns, and you can use them with a wide variety of object-producing tools, such as parallelogram and triangle tools. You can alter the way any tool works to get some very effective results. For example, you can set any of the tools so the last point of one object will be the first point in the next object you create; in this way, the objects you create will be chained together.

Images also has brush effects similar to **Smear**, **Shade**, and **Blend** in Deluxe Paint. "Transparency" and "glow" are two of these. Other features are "pantograph" and "under," which allow you to copy an image with strokes of a brush or to paint one color under other colors already on the screen.

THE IFF GRAPHICS STANDARD

One thing that Deluxe Paint, Graphicraft, and Aegis Images have in common is that they all save their picture files to disk using the IFF graphics standard. IFF stands for Interchange File Format. It's a standard that was developed by Electronic Arts in collaboration with Commodore-Amiga, to make sure that files saved by one program can be used by another program. There are IFF standards for graphics, for music, for sound, for text, and for a wide range of other types of data that can be saved to disk or transmitted. IFF standards are designed to go beyond working with different programs on the same computer—they're also designed to work with different computers.

Software developers have to use the IFF standard, or their programs won't be able to use files from other programs that were saved using the IFF standard. Fortunately, most software developers for the Amiga are adhering to the standard at Commodore-Amiga's urging. What this means for you is that you can take a picture saved by Graphicraft, Deluxe Paint, or Aegis Images and load it into any Amiga graphics program that uses the IFF standard. For example, if you like some features of Deluxe Paint and other features of Images, you can create a picture on Deluxe Paint, save it, and then load it into Images for some touch-up work using Images' special features.

GRAPHICS HARDWARE FOR THE AMIGA

If you're using your Amiga extensively for graphics, then it's worthwhile to make sure you're using hardware that brings out the best in the Amiga's graphic capabilities. The following sections list types of graphics hardware for your Amiga, along with some recommendations to help you choose the equipment that's right for you.

MONITORS

The piece of graphics hardware you probably spend the most time with is the monitor you use with your Amiga. If you're using a monochrome monitor or a composite video monitor and want to upgrade your picture by buying an RGB monitor, there are several things to consider as you shop.

There are two types of RGB monitors: analog and digital. Digital RGB monitors are the type most commonly used with the IBM PC, so they're easy to find. Your Amiga can display pictures on a digital RGB monitor, but a digital RGB monitor can only display 16 different colors, severely limiting the colors you can use with your Amiga. An analog RGB monitor can display a full range of colors—it's the best RGB monitor for working with color graphics.

Many different analog RGB monitors are available. To make sure you find one that works well with your Amiga, be sure that the monitor has at least 400 lines of vertical resolution to display the Amiga's high-resolution pictures with accuracy. You should also check the dot pitch of the monitor; the smaller the dot pitch, the clearer the picture on the screen. An average RGB monitor might have a dot pitch of around .4 mm. An exceptionally clear monitor might have a dot pitch of around .28 mm.

If your budget precludes spending a good chunk of money on a monitor just for your computer, you might consider getting a TV set that will display analog RGB signals from your Amiga. That way, when you're finished with your computer work, you can watch your favorite TV programs. One example of this type of TV/monitor is the Sony KV1311.

Another alternative is to buy any of the many video monitors on the market designed to work with a component video system that also includes an input for an analog RGB signal. They usually don't have a tuner built into them, but most will accept a signal from a VCR, so you can use the tuner built into your VCR to receive television programs. One advantage to a video monitor like this is size: Some of them have a screen size as large as 25 or 26 inches, a real advantage if you're displaying Amiga images in a storefront, or even if you just like to sit back in a sofa and play larger-than-life video games.

PRINTERS

A color printer is very important if you want to make copies of your Amiga images on paper. The Amiga's system software has printer drivers for three different color printers: the Okimate 20, the Epson JX-80, and the Diablo C-150. You can use other color printers, but you must have a printer driver for them before they will work with the Amiga.

The Okimate 20 printer

The Okimate 20 printer, manufactured by Okidata, is a low-cost color printer. It's a thermal dot-matrix printer: It uses a colored waxed ribbon and a thermal print head to burn the color from the ribbon onto the paper. As a result, the colors are well saturated (that is, they look bold, not faint).

There are several drawbacks to the Okimate 20. It uses a ribbon quickly, since it can only print the length of a ribbon once, and then the ribbon must be replaced. The registration of the print head isn't always exact; the pictures it creates sometimes have horizontal bands of white separating each pass of the print head in the picture. Nevertheless, for a printer as inexpensive as the Okimate 20, it's a good value for the money.

The Epson JX-80 printer

The Epson JX-80 is an impact dot-matrix printer that uses a multicolored inked ribbon to strike against paper to print colors. The JX-80 is probably the best text printer of the three color printers, since it can produce near letter-quality text if you add a special chip to it. It can use almost any kind of paper you can feed into it, and it will use one ribbon over and over again until it runs out of ink.

The main disadvantage of a JX-80, like most color impact dot-matrix printers, is that the color saturation you get from an inked ribbon on standard paper is poor; the printed images look washed out, compared to what you see on the screen. The advantage is that you can use the JX-80 for word processing as well as printing graphics, and it's affordable.

The Diablo C-150 and Xerox 4020 printers

If you're serious about printing your color graphics on paper, you should consider using a Diablo C-150 inkjet printer, manufactured by Xerox, or the Xerox 4020 printer, an improved color inkjet printer that uses the C-150 printer driver. You do have to be serious about it, because these printers are relatively expensive. What do you get for the money? Extremely vivid colors and detailed accuracy.

The inkjet printers spray colored ink on a special clay-coated paper that doesn't absorb and spread ink. Since the ink dots are tiny, you get very fine resolution. The inks are comparatively thick and don't sink into the paper, so the printed colors are very well saturated. You can also use clear acetate in these inkjet printers to create colored transparencies, so you can use an Amiga graphics program to create and print business graphics for a presentation on an overhead projector.

These two inkjet printers aren't for everyone. They really aren't practical for word processing, because even though they can produce some beautiful-looking characters, you can print them only on clay-coated paper or acetate, not great media for business letters or other correspondence. The C-150 takes some maintenance to keep it producing clean color pictures; you have to rinse out the inkjet nozzles and occasionally clear trapped bubbles from the ink lines. The 4020 is much easier to maintain, and also produces color with greater saturation than the C-150. Despite the maintenance and cost considerations, these inkjet printers are the best printers to have if you plan to use your color printouts for more than curiosities.

THE AMIGA LIVE! FRAMEGRABBER


If you want to add some pictures from real life to your collection of Amiga images, you can use the Amiga LIVE! framegrabber to turn images from a video camera, TV, VCR, another computer, or any other video source into a low-resolution Amiga image.

The Amiga LIVE! framegrabber plugs into the expansion bus on the right side of the Amiga. On the side of the Amiga LIVE! box is an NTSC jack where you can plug in a standard video cable to carry a picture from another video source (like a video camera). When you turn on the Amiga, you boot a disk containing the Amiga LIVE! driver, the software that reads the images from the framegrabber. When the driver is loaded, you see the images that the framegrabber produces on your monitor screen.

Amiga LIVE! is a real-time framegrabber. That means that at least 12 times a second it can take a video picture and convert it to a 320-by-200 Amiga picture. If the source is a moving picture, you see the result as a series of moving Amiga pictures on the screen. You can at any point freeze the picture on the screen. Amiga LIVE! can capture pictures from video in color using 32 colors, or in black and white using 16 levels of gray. You can then save those pictures to disk, and since the software will save them using the IFF standard, you can load your picture into an IFF graphics program and alter them later.

Amiga LIVE! can act as more than just a still-picture digitizer. Since it operates in real time, and since the Amiga simultaneously puts its images out through the NTSC port on the back of the Amiga console, you can feed the digitized images from Amiga LIVE! into a VCR connected to the NTSC port to create your own special-effects videos. Amiga LIVE! acts as an image processor to freeze frames, alter colors, and create many of the special effects you see in other video-based productions such as music videos.

You've now been able to try some of Deluxe Paint's advanced features, and you've learned how to print and photograph your creations. You've also had a look at some of the other graphics programs on the market for the Amiga. If you're a BASIC programmer, or if you're curious about some of the more technical aspects of Amiga graphics, read on through the next three chapters on Amiga BASIC graphic statements.



**CHAPTER FOUR
AMIGA BASIC
GRAPHICS:
SCREENS,
WINDOWS, AND
PALETTES**

Microsoft's Amiga BASIC has a full set of graphics commands that let you design effective pictures with your BASIC programs. You can use the graphics commands to create your own screens and windows, then fill those windows with shapes, colors, and text. You can display data with colors and shapes instead of numbers, ask for input from the user in visually irresistible ways, and add visual pizzazz just for fun.

In the next three chapters, you'll learn how to use the Amiga BASIC graphics commands. This chapter concentrates on the commands you need to create your own windows, screens, and color palettes. Chapter 5 explains how to draw lines and create shapes in different colors. Chapter 6 shows you how to add text to your graphics, make your graphics adapt to changing window sizes, and cut and paste sections of graphics.

AN INTRODUCTION TO AMIGA BASIC

If you haven't used Amiga BASIC before, you might find that its unique features take a little getting used to. Unlike most versions of BASIC, Amiga BASIC doesn't need line numbers. Another difference is that it uses two different windows for separate activities—one window to enter the program, the other to display the results. To help you use the program examples in the following chapters, this short introduction shows you how to enter, run, and stop Amiga BASIC programs. The tutorial is not intended to teach you BASIC programming—that's a large job best left to a beginning BASIC book or a teacher—but it will help start you out painlessly. If you want more specific information about using Amiga BASIC, you can find it in the user's manual that comes with the program.

THE LIST AND OUTPUT WINDOWS

To start Amiga BASIC, you open the Amiga BASIC icon just as you would any other icon: You point to it with the pointer and double-click the Select button on the mouse. After a few seconds, two windows appear on the screen: the List window and the Output window, shown in Figure 4-1. The List window is titled LIST, and the Output window is titled (for now at least) BASIC. The Output window's title bar shows the title of the program currently in memory. Since you have no program loaded now, it displays BASIC. As soon as you load a program from a disk or save a new program under any name, the Output window's title bar will show the name of the program.

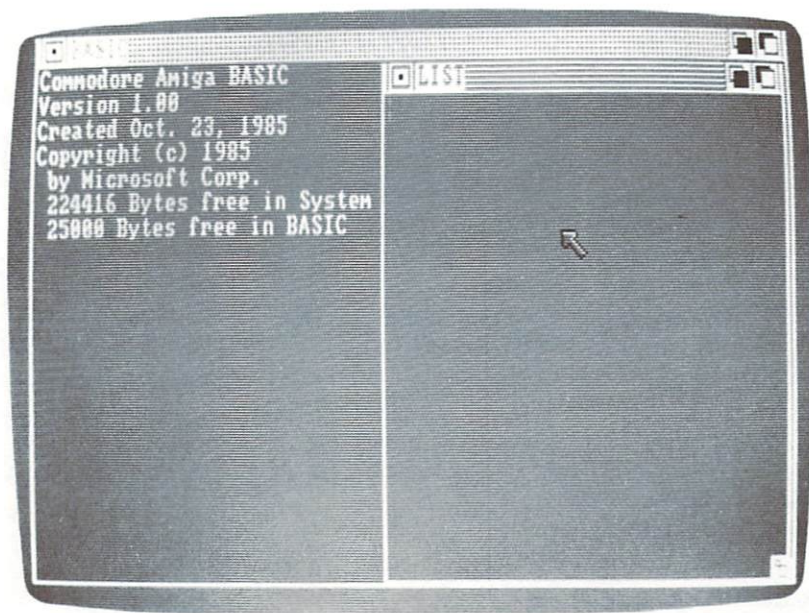


Figure 4-1.

The Amiga BASIC List and Output windows.

These two windows keep the lines of the BASIC program and the results of the program separate. You enter and store the program lines in the List window. When you run the program, any visible results that occur, such as text printouts or graphics, usually appear in the Output window.

ENTERING A PROGRAM IN THE LIST WINDOW

To enter a program in the List window, you must select the window by moving the pointer inside it and clicking the Select button. A vertical cursor line then appears at the upper left corner of the window. Enter the program by typing it in line by line, and pressing the RETURN key at the end of each line. Since you don't need to use line numbers, it's important to keep the lines in the correct order.

If you come to the right edge of the List window before you finish entering a long line, the contents of the window will automatically scroll left to let you continue entering the line. As soon as you press RETURN, the contents will scroll back to the left edge of the program lines. The contents of the window will likewise scroll up when you reach the bottom of the window.

You can correct mistakes and make changes in the program by moving the cursor back to the lines you've already entered. The four cursor keys move the cursor over the program listing without altering any characters underneath it. You can also use the mouse pointer to move the cursor by simply pointing to the spot where you'd like the cursor to appear and clicking the Select button. When

you get to the text you'd like to change, just start typing to insert new text, or use the BACKSPACE key to erase characters before the cursor. You can also delete text by dragging over it with the mouse to highlight it, then pressing the BACKSPACE key.

SCROLLING THROUGH A PROGRAM LISTING

If your program has more lines than you can view in the List window at one time, you can scroll through the listing a page at a time by holding down one of the SHIFT keys and pressing the up or down cursor key, or you can jump to the beginning or end of the program by holding down one of the ALT keys and pressing the up or down cursor key. If you can't see the entire length of the program lines, you can make the List window wider by dragging it left, then using the sizing gadget in the lower right corner to stretch it out to the right. If you still can't see the full length of the lines, use SHIFT with the left or right cursor key to scroll left or right a full window at a time, or ALT with the left or right cursor key to scroll to the end or beginning of a line.

RUNNING A PROGRAM

Once you've entered a program, you can run it by choosing **Start** from the **Run** menu. The List window disappears, and the results appear in the Output window (if they're text or graphics). When the program is finished, the List window reappears. If the program doesn't stop by itself, you can stop it by choosing **Stop** from the **Run** menu, or by pressing the right Amiga key together with the period (.) key, or by holding down CTRL and pressing C. If the List window doesn't automatically appear after the program stops running, you can choose **Show List** from the **Windows** menu, or use the keyboard shortcut and press the right Amiga key together with the L key to make it reappear.

ENTERING IMMEDIATE COMMANDS IN THE OUTPUT WINDOW

If you select the Output window by clicking in it, you can also enter Amiga BASIC statements there. Instead of storing the statements as it does in the List window, Amiga BASIC executes each statement line that you enter in the Output window as soon as you press the RETURN key. This makes each statement an *immediate* statement, since you see its results immediately. You can use the Output window to test the effects of individual commands before putting them in a program.

SAVING A PROGRAM

If you want to save a program that you've written, just choose **Save** from the **Project** menu. Amiga BASIC will prompt you for a name. Click in the box to get a cursor, then type the name in and

press RETURN; your program will be saved on disk as a BASIC file. You'll see the title of the program appear in the title bar of the Output window. As you modify the program, you can continue to use **Save**. Each time you choose it, Amiga BASIC will save the program in its current state under the same name, overwriting the old version already on the disk. To save an existing program under a different name, choose **Save As** from the **Project** menu and type in another name. The name displayed in the Output window's title bar will change to the new name, and the old file will remain untouched on disk.

To load a program from disk, choose **Open** from the **Project** menu, click in the box, then enter the name of the program and press RETURN. If the program you're working on hasn't been saved in its current state, you will be warned and given the opportunity to either save or throw out the changes. After that, Amiga BASIC will erase the program that is currently in the List window and replace it with the program you chose. The same is true if you choose **New** to begin entering a program from scratch.

LEAVING BASIC

When you want to leave BASIC to get back to the Workbench, you can quit Amiga BASIC by choosing **Quit** from the **Project** menu, or by closing both the List and Output windows by selecting their close gadgets. If you just want to see the Workbench without quitting BASIC (to see how full the disk is, for example), you can select the back gadgets on both the List and Output windows to put them behind the other windows on the Workbench. Once you've seen what you want to see, you can bring the List and Output windows back to the foreground by selecting their front gadgets, or by selecting the back gadgets on the other Workbench windows. In some cases, you may have to choose **Show List** from the **Windows** menu to see the List window again.

AN INTRODUCTION TO SCREENS, WINDOWS, AND PALETTES

As you may recall from Chapter 2, the Amiga creates its graphics using pixels, or tiny "picture elements." If you look closely at any picture or text on the Amiga's monitor, you can see the pixels: They're the small, squarish building blocks that make up the images. To build an object on the monitor screen, the Amiga sets the color of individual pixels in the shape of the object. When you see those pixels from a distance of more than a few inches, they combine to appear as one object.

The size of the pixels the Amiga uses to build images determines the resolution of the picture. In a high-resolution picture, the pixels are small and not easily seen, so you can create curves and diagonal lines that appear fairly smooth. In a low-resolution picture, the pixels are larger and more evident; diagonals and curves have a distinct jagged look, and you can see the individual pixels more easily.

SCREENS

The Amiga offers four different resolution modes for displaying graphics and text on the monitor. You choose any of the resolutions and work with it by creating a screen (explained in the SCREEN statement section in this chapter). You can have up to five screens on the monitor at one time, and each can have its own resolution mode. Each Amiga screen extends across the full width of the monitor screen and has a title bar at the top (you can create a screen that is not displayed across the full width, but it will "command" the entire width of the screen). You can grab the title bar with the pointer and drag the screen up and down on the monitor by holding down the Select button of the mouse and rolling the mouse forward and back. This reveals the screen immediately behind the current screen (if any), and you can then grab that screen's title bar and move it up and down.

The screen's title bar is also used to display menus. When you press and hold down the Menu button of the mouse, the titles of the menus appear in the title bar. You can make an individual menu appear by pointing to the menu title with the pointer while the Menu button is still depressed.

The title bar also includes back and front gadgets in the right corner. If there is more than one screen on the monitor at one time, those screens are layered, one covering another. Selecting the left (back) gadget moves that screen behind any other screens, and selecting the right (front) gadget moves that screen in front of any screens on the monitor.

A good example of a screen is the Workbench screen. When you first boot the Amiga, the Workbench screen appears with **Workbench release 1.1** (or something similar) in the title bar and the icons for any disks you have in the drives. You can drag the screen up and down

by pointing to the title bar with the pointer, holding down the Select button, and rolling the mouse forward and back. If you try the back and front gadgets, nothing happens because there aren't any other screens on the monitor at this point. When you press the Menu button on the mouse, you get the Workbench menus in the title bar. You can see the Workbench screen, complete with gadgets, in Figure 4-2 (on the next page).

WINDOWS

Screens provide areas of different resolution on the monitor. Within each screen, *windows* provide clearly defined areas of activity. If you want to create graphics or put text on a screen, you must do it within a window—you can't put graphics or text on a screen without a window. Windows share the same resolution and colors as the screen they belong to, so you can't create windows with different resolutions or colors on the same screen.

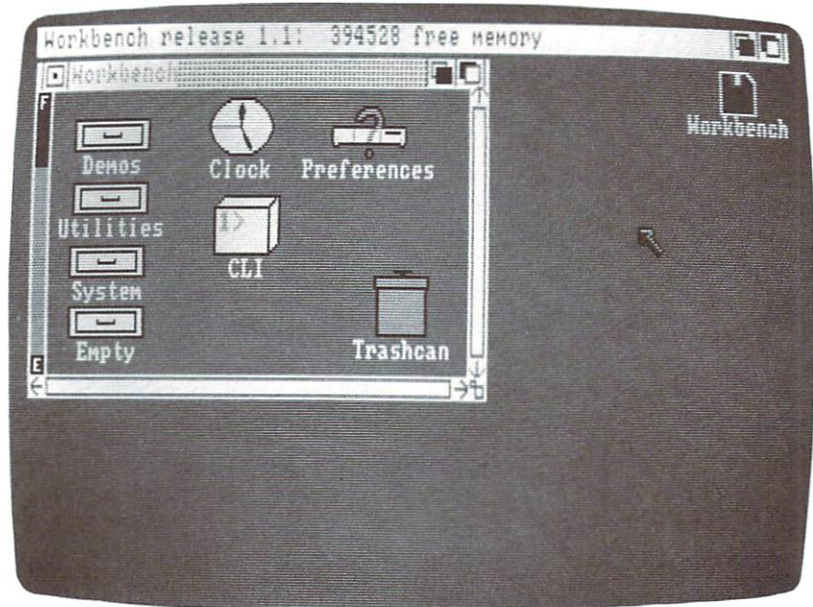
Unlike screens, windows don't have to extend across the full width of the monitor. They can be almost any size, as long as they fit within the screen they're on. Each window has its own boundaries, and can include an optional title bar at the top. The title bar can show the name of the window, and can also include three optional gadgets: the close gadget, the back gadget, and the front gadget. Selecting the close gadget will make the window disappear from the screen. Selecting the back gadget will move the window behind any other windows on the screen; selecting the front gadget will move the window in front of any other windows on the screen.

In the lower right corner of the window is an optional sizing gadget. By pointing to the sizing gadget, holding down the Select button on the mouse, and then rolling the mouse, you can change the size of the window. Also, if the window has a title bar and is smaller than the screen, you can move the entire window around within the boundaries of the screen by pointing to the window's title bar, holding down the Select button, and rolling the mouse.

If you open the Workbench disk icon on the Workbench screen as shown in Figure 4-2, you can see a good example of a window with all the gadgets, the title bar, and full mobility. You can drag it around to any spot on the screen, resize it, and close it if you wish. If you drag the Workbench screen up and down on the monitor, all the windows it contains will move with it.

Figure 4-2.

A window on the Workbench screen showing the contents of the Workbench disk.



PALETTES

Each screen on the Amiga monitor also has its own set of colors, called a *palette*. The palette for each screen can have up to 32 different colors. You can choose each of the colors in the palette from a total of 4096 different colors, so it's possible to create a unique palette for each screen. The procedures for changing the colors in the palette from BASIC are explained later in this chapter.

The Amiga uses the colors in the palette to color all the pixels in its screen and any windows on that screen that use those colors. For example, the first color in the palette is used to color the background of the screen and its windows, and the second color is used to fill in the boundaries of the windows and the title bars. The last two colors in the palette are used to create inverse colors that appear when you select any menu item.

When you first boot the Workbench, you see a blue background with white window borders and title bars (that is, if you haven't changed the colors). When you select menu items, they turn to the inverse colors of black and orange. If you use Preferences to change your Workbench colors, you are choosing new colors for the palette of the Workbench screen, which show up as new colors for the background, window borders, and inverse menu lettering.

THE SCREEN STATEMENT

When you start working with Amiga BASIC, you use the List and Output windows that BASIC creates on the Workbench screen. Any graphics you create or text you print will appear in the Output window using the Workbench screen's resolution and four-color limitation. You'll probably want to use more colors and a different resolution. To do so, you need to create a new screen with the SCREEN statement. It uses this format:

```
SCREEN screen ID number, width, height, depth, resolution mode
```

The five specifications following SCREEN have to appear in this order, and must be separated by commas.

THE SCREEN ID NUMBER

You can create up to four screens in Amiga BASIC. These are in addition to the Workbench screen, so you can have a total of up to five screens on the monitor at once. To identify your screens, you give each screen an ID number of 1, 2, 3, or 4. The Workbench screen is always numbered -1. Later in this chapter, when you use the WINDOW statement to create windows, you'll use the screen ID number to tell BASIC which screen you want to put the window on.

THE SCREEN RESOLUTION MODE

Although the screen resolution mode is the last specification you enter in the SCREEN statement, you need to know what resolution you're going to use before you set the width, height, and depth of the screen. You can choose any one of four different resolution modes. Each mode uses a different size pixel.

There are two different pixel widths: fat, called low-resolution, and skinny, called high-resolution. High-resolution pixels are skinny enough to fit 640 of them across a full screen. Low-resolution pixels are twice as wide as high-resolution pixels; 320 low-resolution pixels fit across a full screen.

There are also two different pixel heights: tall, which is called non-interlaced, and short, which is called interlaced. On a full screen, 200 non-interlaced pixels fit from top to bottom. Interlaced pixels are half as high as non-interlaced pixels; 400 interlaced pixels fit from top to bottom on a full screen.

By combining the two different widths and two different heights of the pixels, you have four different resolutions available that you can specify with the following resolution mode numbers:

- **Mode 1:** Low-resolution, non-interlaced pixels that measure a maximum of 320 by 200 on a full screen.
- **Mode 2:** High-resolution, non-interlaced pixels that measure a maximum of 640 by 200 on a full screen.
- **Mode 3:** Low-resolution, interlaced pixels that measure a maximum of 320 by 400 on a full screen.
- **Mode 4:** High-resolution, interlaced pixels that measure a maximum of 640 by 400 on a full screen.

For reference, the Workbench screen is a mode 2 screen.

Any screens that use mode 3 or 4 flicker slightly on the monitor because they use interlaced pixels. To fit 400 lines of pixels on the monitor, the Amiga uses a display method called interlacing that first draws all the odd-numbered raster lines and then draws all the even-numbered raster lines. It alternates back and forth between the odd and even lines once every $\frac{1}{30}$ of a second, which is slow enough to create a slight flicker. Non-interlaced pixels don't flicker because the Amiga draws all the lines in one pass every $\frac{1}{60}$ of a second, fast enough to avoid a flicker.

Anytime there are several screens on the monitor and any one of them is an interlaced screen, the entire monitor will flicker slightly, since the monitor must use the interlacing display method for its entire screen. The interlacing won't affect the size of the pixels in the non-interlaced screens. Apart from the flicker, they'll look just as they normally do. Closing all interlaced screens will get rid of the flicker on the monitor.

SCREEN WIDTH AND HEIGHT

Once you've decided on a resolution mode, you can set the size of the screen by setting width and height in pixels. To set the width and height, you need to consider the limits of your screen resolution and keep within them. For example, you can't create a mode 1 screen measuring 400 by 200 pixels because a mode 1 screen has a maximum width and height of 320 by 200 pixels.

If you create a screen that is smaller than the maximum dimensions of its resolution, BASIC positions the screen in the lower left corner of the monitor display. For example, a mode 1 screen set to a width and height of 160 by 100 would appear on the monitor as

shown in Figure 4-3. You can't drag this screen above its current position, but you can drag it down lower. Notice also that the area to the right of the screen is blank, but it will drag up and down with the screen, since the resolution of the screen covers the full width of the monitor.

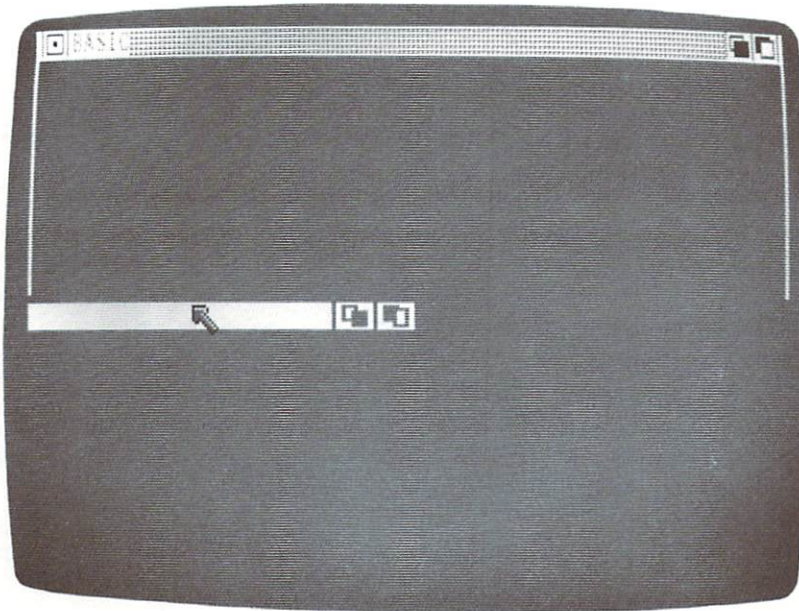


Figure 4-3.

A 160-by-100 mode 1 screen.

SCREEN DEPTH

The depth of a screen is measured in bit planes. The number of bit planes in a screen determines how many colors the screen and all the windows in that screen can display. Each bit plane is a section of the Amiga's RAM that uses one bit to store color information for each pixel in the screen. By combining bit planes, the Amiga can use multiple bits for color storage for each pixel in the screen.

You can specify anywhere from one to five bit planes when you create a screen. A screen with one bit plane supports just two colors, because there's only one bit in memory for each pixel (the bit can be either on or off, hence two colors). Each additional bit plane you add to a screen assigns one additional bit in memory for each pixel, allowing more colors to choose from for displaying each pixel. For example, a 2-bit-plane screen assigns two bits to each pixel, giving you four colors to choose from—since two bits allows four combinations of zeros and ones. Each bit plane you add doubles the color capacity of each pixel, since each bit doubles the possible combinations of zeros and ones. The following chart

shows how many colors are available for each pixel when you assign the corresponding screen depth:

Number of bit planes	Number of colors
1	2
2	4
3	8
4	16
5	32

Although you can specify up to five bit planes in resolution modes 1 and 3, you are limited to four bit planes (16 colors) in modes 2 and 4.

SCREEN MEMORY REQUIREMENTS

When you set a screen's dimensions, BASIC reserves enough RAM to contain the bit planes for that screen. To get an estimate of how much memory each screen takes, multiply the three screen dimensions together to get the number of bits, then divide by eight to get the number of bytes:

$$\text{height} \times \text{width} \times \text{depth} \div 8 = \\ \text{approximate screen RAM in bytes}$$

For example, a 320-by-200-by-5 screen would use approximately 40,000 bytes of memory (320 times 200 times 5 divided by 8).

Amiga BASIC reserves RAM for the screens in the system memory (not in the original 25,000 bytes set aside by BASIC to store programs). When you create screens, you should keep in mind how much system memory you have available, and be careful not to overload it. The best way to conserve memory is to limit the depth of your screen if you don't need many colors. A 320-by-200-by-5 screen can use 32 different colors, since it uses 5 bit planes. If you cut it down to 4 colors, requiring just 2 bit planes, you'd need only 16,000 bytes of RAM for the screen instead of 40,000.

SCREEN STATEMENT EXAMPLES

Once you've decided on a screen-resolution mode, the dimensions of the screen, and the screen ID number, it's very easy to create a screen. As a simple example, to create a mode 4 screen with the screen ID number 1 that measures 640 by 400 with 2 bit planes, you'd enter:

```
SCREEN 1, 640, 400, 2, 4
```

To see how to create several screens of different resolutions at once on the monitor, enter and run this very short program:

```
SCREEN 1, 320, 200, 1, 1
SCREEN 2, 640, 200, 1, 2
SCREEN 3, 320, 400, 1, 3
SCREEN 4, 640, 400, 1, 4
```

This program creates four full-size screens, one in each resolution, each with a depth of only one bit plane (to conserve memory). You won't see the screens immediately, because they disappear behind the Workbench screen as soon as the program stops running (which is almost immediately). To see them, you need to first resize both the BASIC Output and List windows and drag them to the bottom of the Workbench screen so you can see the Workbench title bar. Then, to see the screen behind the Workbench screen, you need to drag the entire Workbench screen down by its title bar. You should see the title bar of screen number 4. If you drag screen 4 down, you can see screen number 3. Dragging screen 3 down reveals screen 2, and dragging screen 2 down reveals screen number 1, so you can see five different screens on the monitor at once, as shown in Figure 4-4. You can use the back and front gadgets on the screens to move them to the front or hide them in the back if you want.

Mode 1 (320 x 200) Mode 2 (640 x 200) Mode 3 (320 x 400) Mode 4 (640 x 400)

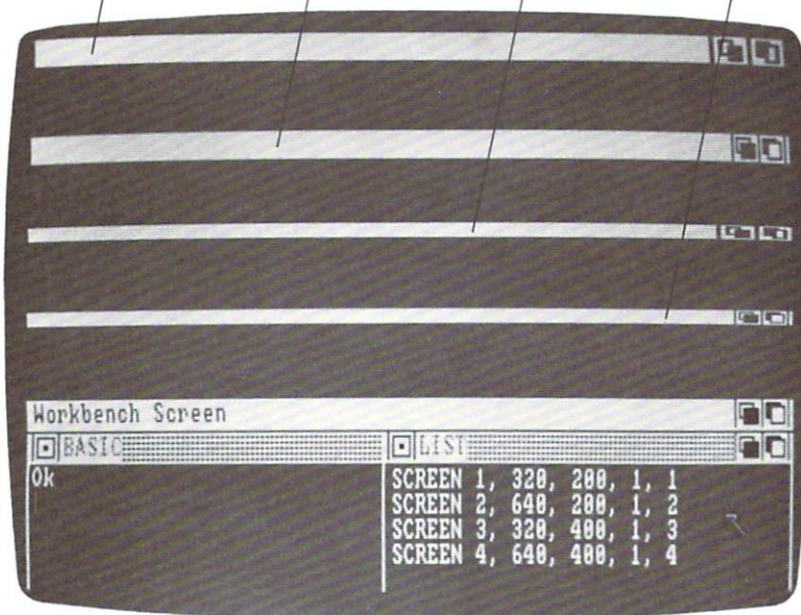


Figure 4-4.

Five different screens on the monitor at one time.

In Figure 4-4, you'll notice that the sizes of the title bars and gadgets change from screen to screen, depending on the screen's resolution. You can also see that the screens are layered in the order you created them. Each new screen you create always appears in front of any screens you previously created.

THE SCREEN CLOSE STATEMENT

When you're finished with a screen, you should close it to make it disappear from the monitor and to release the RAM the Amiga uses to store the screen. The SCREEN CLOSE statement will do that. It uses this format:

```
SCREEN CLOSE screen ID number
```

The screen ID number specifies the screen you want to close. For example, to close screen 3, you'd use:

```
SCREEN CLOSE 3
```

To return your monitor screen back to normal after running the previous multiple-screen program, drag the title bar of the Workbench screen back up to the top of the monitor, click in the BASIC Output window to select it, and then enter these four immediate commands:

```
SCREEN CLOSE 4  
SCREEN CLOSE 3  
SCREEN CLOSE 2  
SCREEN CLOSE 1
```

You should notice that the interlacing flicker disappears as soon as you close screens 3 and 4, the interlaced screens.

THE WINDOW STATEMENT

When you create a screen, you set the resolution and number of colors you want to work with. To create an area on that screen to put graphics and text into, you must create a window with the WINDOW statement. You can create as many windows as the

Amiga's memory will support, and you can make them in different sizes, locations, and with different features. You can make windows in the Workbench screen or any other screens you have already created with the SCREEN statement.

The format for the WINDOW statement is:

```
WINDOW window ID number, title, opposing corner addresses,  
window features number, screen ID number
```

Each of the specifications following WINDOW must come in the order shown in the format, and all specifications must be separated by commas. The window ID number must be included in every WINDOW statement. Because the other specifications are not necessarily relevant for every window, they are optional, as you'll see shortly. However, if a specification is not set but specifications after it are set, a comma must still be included at its position.

THE WINDOW ID NUMBER

To identify and keep track of more than one window, you need to assign a window ID number to each window you create. It can be any integer from 1 up to 255. Window ID number 1 is the BASIC Output window, so if you want to create a new window without altering the Output window, you should start with ID number 2 and continue numbering from there with each new WINDOW statement.

THE WINDOW TITLE

The title specification of the WINDOW statement lets you include a name that will appear in a window's title bar. You can use any string you want as a title. For example, you can use a string in quotes like "Aieeeee!" or you can use a string variable like TITLE\$ that's been assigned earlier in the program. It's a good rule of thumb to always end the string with a space to make the title appear cleaner and less cramped in the title bar. If you don't want a title, and you want to include other specifications that are to the right of the title in the WINDOW statement, you must include a comma at the title specification's position. The window's title bar will then be blank.

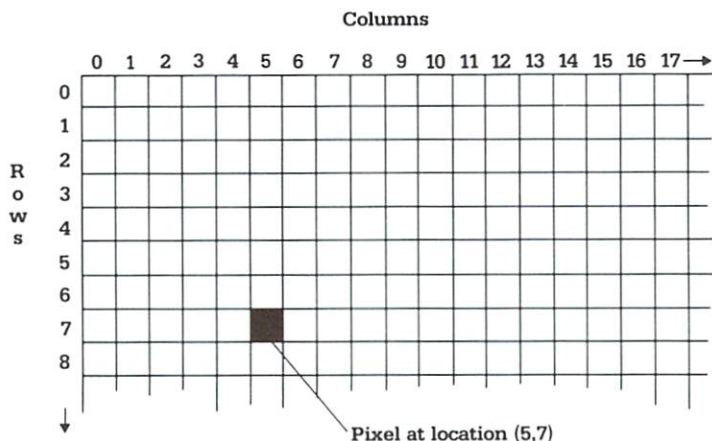
OPPOSING CORNER ADDRESSES

You set the size and location of the window by specifying the location of two of its opposing corners on the screen. To specify the two corners on the screen, you need to use pixel addressing. Pixels are arranged on a screen in a grid, using numbered rows and columns that start in the upper left corner of the screen. The top

row of pixels is row 0, the rows following below it are numbered 1, 2, 3, 4, and so on until you reach the bottom of the screen. The first column of pixels on the left is numbered column 0, and the columns following it to the right are numbered 1, 2, 3, 4, and so on until you reach the right side of the screen. Figure 4-5 shows you how this numbering system works.

Figure 4-5.

Amiga BASIC's pixel-addressing system.



To locate any one pixel, you use a pixel address that is the pixel's column number followed by its row number. For example, a pixel at address 5,7 is 6 columns over and 8 rows down from the upper left corner of the screen. Because the numbering starts with zero, the actual row and column is always one count greater than the row and column number in the address.

In Amiga BASIC, pixel addresses are always enclosed in parentheses with the column number first, followed by a comma and the row number. The pixel address shown before would appear as (5,7) in a BASIC program. When you use two addresses, as you do to show the two opposing corners of a window, you separate them with a dash. So a window occupying the upper left quarter of a 320-by-200 screen would be shown as (0,0) - (159,99). These two addresses are the upper left corner of the screen and the center of the screen. The window they position looks like the one shown in Figure 4-6.

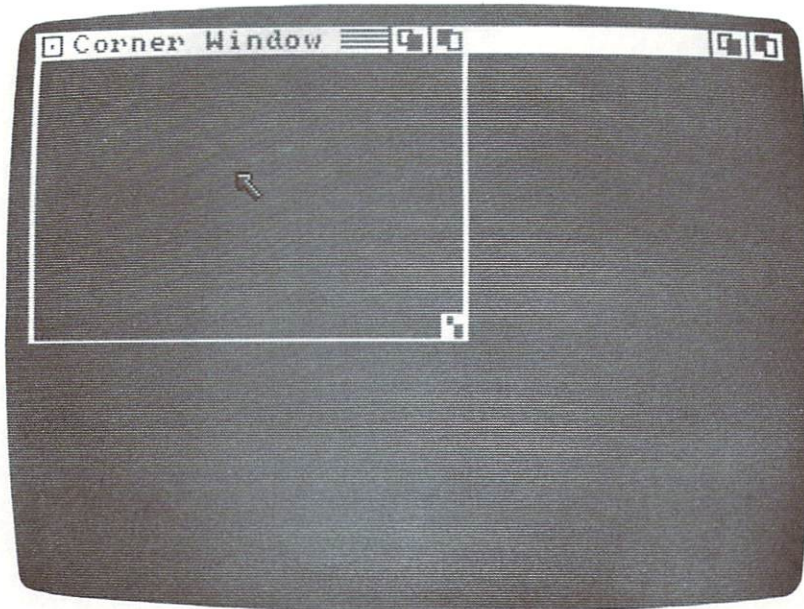


Figure 4-6.

A window created in the upper quarter of a screen.

When you enter opposing corner addresses, make sure your addresses are within the boundaries of the screen. The boundaries are decided by the resolution and the size of the screen you set in the SCREEN statement. If you have a 640 by 400 screen, then (10,20) – (300,450) won't work because the second address goes off the bottom of the screen.

If you decide not to include opposing corner addresses in the WINDOW statement and simply include a comma at their position, the window you create will fill the entire screen.

WINDOW FEATURES

Windows come in different sizes and varieties, as you've probably found out by working with different Amiga programs. Some windows can be dragged around the screen, while others are stationary. Some have all sorts of gadgets; others have none.

The WINDOW statement gives you the chance to customize your window as you create it. You can pick features for the window somewhat like choosing items from a catalog. Each feature has a value associated with it. After you decide which features you want in your window, you take the value for each feature and add all the values together. The resulting number is the window features

number, and it tells WINDOW which features you've chosen. The following is a list of window features you can specify with their corresponding feature values:

- **Sizing gadget (1)**—A sizing gadget appears in the lower right corner of the window. You can drag the sizing gadget with the pointer to change the size of the window. Without a sizing gadget, a window can't be changed from its original size by the user (sometimes a desirable feature in itself).
- **Window dragging (2)**—You can drag the window around the screen by its title bar, using the pointer. Without window dragging, a window remains fixed where it was created.
- **Back and front gadgets (4)**—Back and front gadgets appear in the upper right corner of the window. You can select either gadget to make the window move to the front or hide behind other windows.
- **Close gadget (8)**—A close gadget appears in the upper left corner of the window. By selecting the close gadget with the pointer, you can close the window, making it disappear from the screen. Without a close gadget, you can't remove a window from the screen with the pointer.
- **Window refreshing (16)**—The contents of the window are redrawn (refreshed) if they've been covered by another window and then exposed again, or if the window has been resized much smaller and then enlarged again. Without window refreshing, any part of the inside of a window will be erased for good if it's covered with another window or if it disappears when the window is resized.

Figure 4-7 shows a window with all of the features that are visible on a window. (It obviously can't show window dragging or refreshing!)

As an example of specifying window features, if you want a window with a sizing gadget, back and front gadgets, and a close gadget, you sum their feature values—1, 4, and 8—to get 13. Entering 13 as the window-features number after the WINDOW statement specifies those features. If you don't enter any window-features number and simply include a comma where it would be, WINDOW assumes you want all the features.

Not all of these features are free. Some will cost you extra memory, and you should avoid specifying them unless you really need them. Window refreshing takes a good-sized chunk of RAM to store the contents of the window. The amount depends on the size of the window; a large window needs much more RAM for window

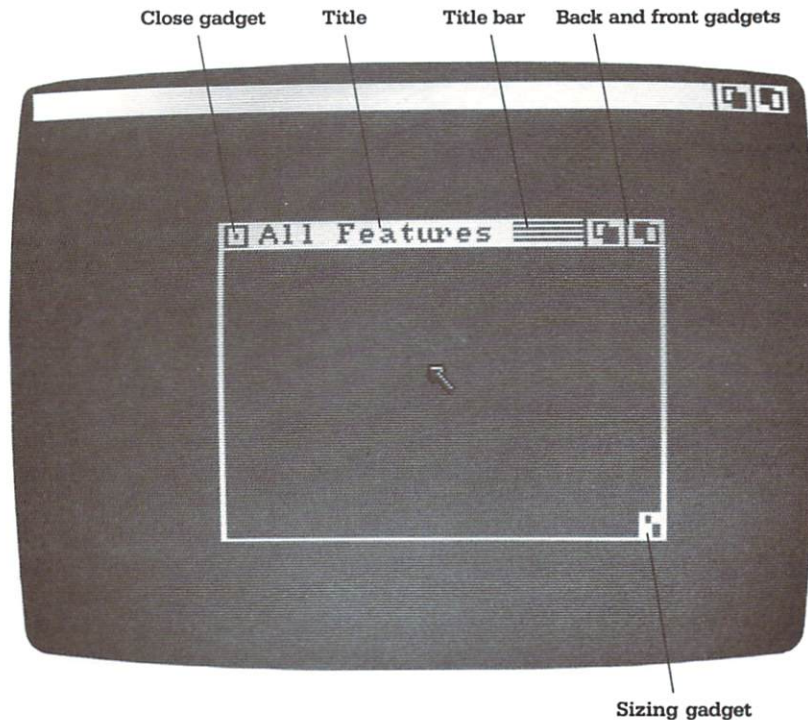


Figure 4-7.

An Amiga BASIC window with all possible features.

refreshing than a small window. Also, a window with a sizing gadget needs the same amount of RAM as a full-screen window, regardless of its initial size, since it can be resized to a very large window at any time.

One feature that's not mentioned directly on the features list is the title bar. If you choose back and front gadgets, window dragging, or a close gadget, or if you enter a title, your window will automatically have a title bar. If you don't choose any of these options, your window will have no title bar.

THE SCREEN ID NUMBER

The last specification you have to include in the WINDOW statement is the screen ID number. You use this number to tell the WINDOW command which screen you want to put your window on. This is important, because the window will use the resolution and color palette of the screen it's on. The screen ID number can be -1, 1, 2, 3, or 4, and should correspond to the screen ID number you set when you created a screen with the SCREEN statement. You can't specify the screen ID number of a non-existent screen; if you ask WINDOW to create a window on screen number 2 and there is no screen number 2, BASIC will notify you of an error.

If you want to create a window on the Workbench screen, specify screen number -1, the Workbench screen's ID number. If

you don't specify a screen number, BASIC assumes you mean the Workbench screen, and will automatically create your window there.

WINDOW STATEMENT EXAMPLES

To give you an idea of how windows are created, consider creating a window titled "Dirty Window" on screen number 1, which has two bit planes, and is a mode 1 screen with dimensions of 320 by 200. This window will be window number 2, and will appear in the middle of the screen. It has all the window features except window refreshing. The WINDOW statement to create it will be:

```
WINDOW 2, "Dirty Window ", (79,49) - (239,149), 15, 1
```

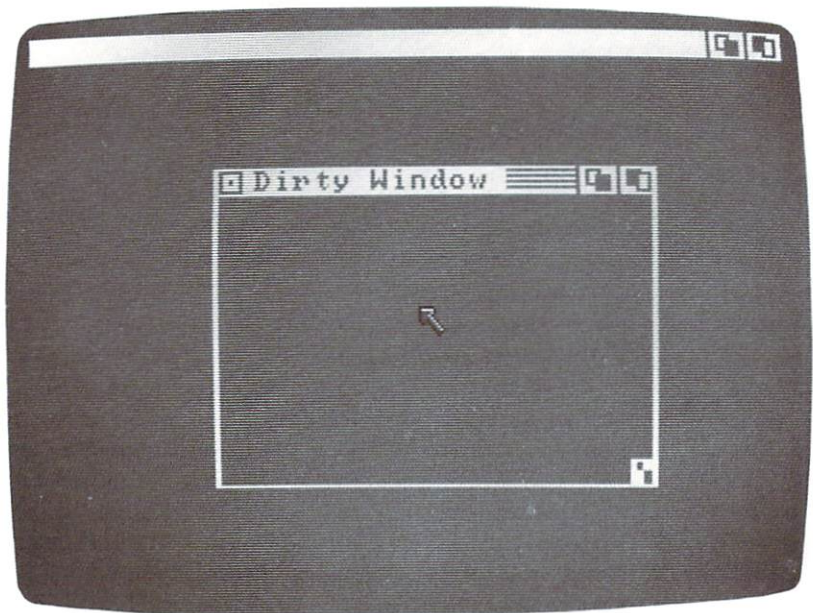
If you want to try it out, enter and run this short program:

```
SCREEN 1, 320, 200, 2, 1  
WINDOW 2, "Dirty Window ", (79,49) - (239,149), 15, 1
```

The screen created by this program should look like Figure 4-8. You can move the window around with the pointer and use the gadgets. To return to the Workbench screen, choose **Show List** from the **Windows** menu. The List and Output windows will then reappear.

Figure 4-8.

The window created by the "Dirty Window" program.



As mentioned earlier, the only specification that has to be in the WINDOW statement is the window ID number. The statement

```
WINDOW 2
```

will work just fine. It creates window number 2, which has all the window features, no title, and fills the entire Workbench screen.

If you want to include a specification at the end of the statement without filling in all the other specifications, you must insert commas in place of the omitted specifications, or you'll get a syntax error. To open a window number 2 in screen number 1 with no other specifications, you'd use the command:

```
WINDOW 2, , , , 1
```

THE OUTPUT WINDOW

Even if there is more than one window open, Amiga BASIC will create graphics and print text in only one window at a time. If that weren't the case, BASIC wouldn't know which window to use to display the results of any graphics or printing commands you issue. The one window that is currently active for text and graphics is called the output window.

When you create windows with the WINDOW statement, the last window you create is automatically set to be the output window. All output from graphics commands and text from print statements that follow will show up in that window.

You can use the WINDOW statement to re-create a window, bringing it up in front of other windows and making it the output window. For example, the short program shown below will create two windows, 2 and 3, and then re-create 2, bringing it out from behind 3 in the same location and size, making it the output window. The PRINT statement at the end of the program tests to see which window becomes the output window.

```
WINDOW 2, "Window 2 ", (0,0) - (250,100)
WINDOW 3, "Window 3 ", (200,0) - (400,100)
WINDOW 2
PRINT "This is the output window."
```

Notice that the title bar of the output window is ghosted (dimmed). Only the title bar of the input window (the window where you put the pointer and click the Select button) will have an

unghosted title bar. The input and output windows don't necessarily have anything to do with each other.

Notice also that the third program line recreates Window 2 without changing its original size and location. Whenever you use a WINDOW statement without specifications for a window that already exists, BASIC uses the first set of specifications you chose for that window.

THE WINDOW OUTPUT STATEMENT

If you want to make an existing window the output window without moving it in front of the other windows, you can use the WINDOW OUTPUT statement, which uses this format:

```
WINDOW OUTPUT window ID number
```

You specify the ID number of the existing window you want to turn into the output window. After BASIC executes the WINDOW OUTPUT command, the window you specified will now display text and graphics without moving in front of the other windows. If you substitute the following two lines for the last line in the previous program example, window 3 will become the output window while it remains behind window 2. The PRINT command will print a message in window 3 so you can see it print out text:

```
WINDOW OUTPUT 3  
PRINT "Hello there, window peeker."
```

THE WINDOW CLOSE STATEMENT

As you create more windows on a screen, you'll notice that it takes longer to move and resize the windows with the pointer, because the Amiga is slowed down by the number of windows it has to keep track of. It's important to close any windows you aren't using to free up the RAM they use so that the Amiga can work at top speed.

You can close a window in two ways: If it has a close gadget, you can select the close gadget with the pointer and the window will disappear. If you want to close the window from within a

BASIC program, you can use the WINDOW CLOSE command, which uses this format:

```
WINDOW CLOSE window ID number
```

The window ID number specifies which window you want to close. For example, this command closes window number 2:

```
WINDOW CLOSE 2
```

PALETTES AND COLOR REGISTERS

As you recall, each screen on the monitor has its own palette with its own array of colors that it uses to color in its background, borders, menu items, and any graphics and text in any of its windows. The Amiga uses 32 different locations in RAM, called color registers, to store those colors. The color registers are numbered from 0 to 31, and there is a separate set of 32 registers for each screen.

When you first turn on the Amiga, the color registers are already filled with default colors that you can use without changing if they suit your needs. If you'd like to see them, run this program:

```
SCREEN 1, 320, 200, 5, 1
WINDOW 2, "Color Registers ", , , 1
PSET(0,190)
FOR i = 0 TO 31
    LINE STEP(0,-180) - STEP(9,180), i, BF
NEXT i
```

This program draws an array of vertical color bars across the screen. (Don't worry about the PSET and LINE statements yet—you'll learn about them in the next chapter.) The color bars show the contents of the color registers in order from 0 to 31, going from left to right. (If you count only 31 colors, that's because the bar on the far left is colored the same as the background color.)

The Amiga follows predetermined rules to use its color registers: Color register 0 is always the background color, and color 1 is the foreground color the Amiga uses to print its text, window frames, and other standard Workbench features. Color registers 2 and 3 are the colors used for inverse-video menu items on the Workbench. Color registers 17 to 20 are the colors used for the pointer. (The pointer is not considered part of the screen, so it can use these color registers even though the screen can't.) If you use Preferences

in the Workbench to change your Workbench and pointer colors, you change the contents of color registers 0 to 3 and 17 to 20.

You can't always use all 32 color registers in each screen. The number of color registers you can use in any screen is limited by the screen's depth in bit planes. The following chart shows which color registers are used for screens of different depths:

Number of bit planes	Color registers used
1	0 and 1
2	0 to 3
3	0 to 7
4	0 to 15
5	0 to 31

THE COLOR STATEMENT

Most Amiga BASIC drawing statements create their figures with the foreground color stored in color register 1. Text is also printed in the foreground color. Some drawing statements also use the background color stored in color register 0 to fill in figures. You can change both the foreground and the background colors to different available color registers with the COLOR statement. It uses this format:

```
COLOR foreground color register number, background color register number
```

To set the foreground and background colors, you specify the register number you want to use for each color, separated by a comma. For example, the following command specifies the color in register 2 as the foreground color and the color in register 3 as the background color.

```
COLOR 2, 3
```

If you leave out the comma and the second number, COLOR sets just the foreground color without altering the background color. If you follow COLOR with a comma and then a number, it sets the background color without changing the foreground color. If you

don't use the COLOR statement in a program, BASIC assumes that the color in color register 1 is the foreground color and the color in register 0 is the background color. Once you set new foreground and background colors with the COLOR statement, BASIC uses them for any drawing commands that follow. By changing the background and foreground colors between drawing commands, you can set new colors for new figures.

Changing the foreground color will have an obvious effect: Any drawing or text statements that follow the COLOR statement will use the new foreground color, since BASIC statements that create graphics draw with the foreground color, as you'll see in the next chapter. Changing the background color is not always so obvious. Unless you use a statement that uses the background color when it executes, you won't see any change to the current background color. The important thing to remember about changing the foreground and background colors is that they will affect drawing statements that follow the COLOR statement—anything created up to that point will remain on the screen in the color it was created with (unless you re-create it after the subsequent COLOR statement).

When you choose foreground and background colors with the COLOR statement, you're restricted to the colors available in the color registers that the screen contains. For example, if you're drawing in a screen with two bit planes, that screen has four color registers and your choice is limited to just those four colors. If the default colors in the four color registers aren't to your liking, you'll want to change them to colors that you like so the choice, although limited, offers you desirable colors. The PALETTE statement lets you change the contents of any color register to any one of 4096 different colors.

THE PALETTE STATEMENT

The PALETTE statement uses this format:

```
PALETTE color register number, red value, green value,  
blue value
```

The color-register number, an integer from 0 to 31, specifies the color register you want to change. The red, green, and blue values are fractional numbers from 0 to 1 that specify the relative strengths of red, green, and blue you want to mix together to create a new color. To understand how these strengths are set, you must know how a color register stores a color.

HOW COLOR REGISTERS STORE COLOR

Each color register stores one color as a combination of the primary colors—red, green, and blue. When you tell BASIC what color you want to store in the register you specify, you must tell it the strength of each primary color that you want to use to make up the new color. There are 16 levels of strength that can be specified for each of the primary colors, making a total of 4096 colors to choose from (16 times 16 times 16 equals 4096).

Instead of specifying whole numbers from 1 to 16 (for example, 5 parts red, 8 parts green, and 16 parts blue), Amiga BASIC requires you to specify the strength of each color as a fractional number from 0 to 1. A value of 0 means that the primary color isn't present, while a value of 1 means the color is present in full strength. The chart in Figure 4-9 will help you determine the fractional numbers to use in the PALETTE statement to represent the strength of each primary color. Using this chart, you can specify a color as shown above (5 parts red, 8 parts green, and 16 parts blue), and then translate these numbers into the fractional numbers that BASIC expects in the PALETTE statement (.28, .47, and 1 in our example).

Figure 4-9.

The 16 primary-color strengths and their corresponding PALETTE values.

Primary color strength	PALETTE value
1	0.00
2	.09
3	.15
4	.22
5	.28
6	.34
7	.40
8	.47
9	.53
10	.59
11	.65
12	.72
13	.78
14	.84
15	.90
16	1.00

To set a new color, you decide on the strengths of each of the three primary colors, then enter the number of the color register you want to change, followed by the corresponding values from the

table above, after the PALETTE statement. For example, the following statement will change the color in register 0 (the background color register) to a shade of green by setting red to strength .34, green to .84, and blue to 0 (no blue present):

```
PALETTE 0, .34, .84, 0
```

CREATING DIFFERENT PALETTES FOR DIFFERENT SCREENS

Recall that each Amiga screen has its own set of color registers. As you create new screens with the SCREEN statement, Amiga BASIC simply copies the contents of the color registers in the original Workbench screen to the color registers of the new screens. To create an entirely new set of colors for a new screen, just use a set of PALETTE statements immediately after you use the SCREEN statement—PALETTE affects only the color registers of the last screen you create.

As an example, this short program creates the color bars you created a few pages ago on one screen, then creates an entirely new color palette for a new (second) screen, then displays the results of the new palette:

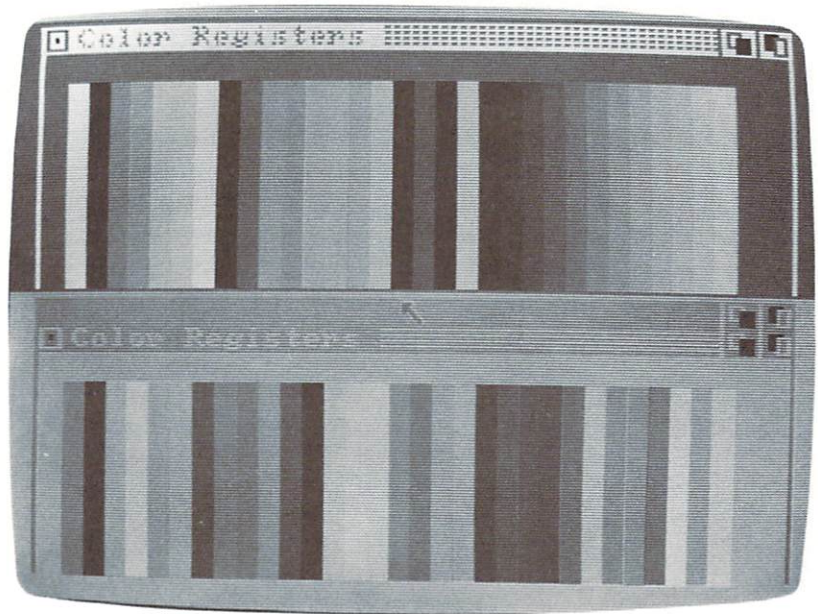
```
SCREEN 1, 320, 200, 5, 1
WINDOW 2, "Color Registers ", , 15, 1
PSET(0,190)
FOR i = 0 TO 31
    LINE STEP(0,-180) - STEP(9,180), i, BF
NEXT i
SCREEN 2, 320, 200, 5, 1
WINDOW 3, "Color Registers ", (0,10) - (297,185), 15, 2
FOR i = 0 TO 31
    PALETTE i, RND(1), RND(1), RND(1)
NEXT i
PSET(0,190)
FOR i = 0 TO 31
    LINE STEP(0,-180) - STEP(9,180), i, BF
NEXT i
```

The loop just after the WINDOW 3 statement uses the PALETTE statement to create a random set of colors for the second screen's color registers. When you run the program, you can see these random colors. If you drag the randomly colored screen down by its title bar, you can see your first screen with an entirely different set of colors just behind it. By putting different screens on the monitor,

each with its own unique set of colors, you can display more than 32 colors on the monitor at one time (shown below in Figure 4-10).

Figure 4-10.

Two different screens with two unique screen palettes.



You've learned how to create screens so you can set the resolution you want to use; how to create windows to set the size of the area in which you put your graphics; and how to create new color palettes so you set precisely the colors you want to work with—everything short of actually creating a picture. These are preliminary activities. Think of them as setting up your canvas and mixing your paints. In the next chapter, you will learn to apply the paint to the canvas and create a picture with the Amiga BASIC drawing commands.



**CHAPTER FIVE
AMIGA BASIC
GRAPHICS:
CREATING IMAGES**

Creating a new screen, setting an output window, and deciding on the colors you want in a palette are only the preliminary steps of creating BASIC graphics on the Amiga. Once you've set up the area you're going to work in, you can start being imaginative, filling windows with figures of your own creation.

In this chapter, you'll learn about Amiga BASIC statements that allow you to draw simple shapes, change the color of a single pixel or an entire window, and create different patterns for drawing lines and filling in figures. You'll also learn a new way to address pixels, and find out how to set background and foreground colors. When you're finished with this chapter, you should be well on your way to becoming an Amiga BASIC artist.

THE GRAPHICS CURSOR AND PIXEL ADDRESSING

Each of the Amiga BASIC drawing statements that you'll use in this chapter draws with an invisible graphics cursor. To tell each drawing statement where to start and end its work, you must understand how the graphics cursor works, and how to use pixel addresses to specify a starting and ending pixel.

THE GRAPHICS CURSOR

When you use Amiga BASIC graphics statements that use pixel addresses, you move an invisible graphics cursor the size of one pixel around the window. This graphics cursor changes the color of the pixels addressed by the drawing statement as it passes over them. If you use a simple statement that alters just one pixel, the graphics cursor moves to the pixel, changes its color, then stays in the same location until you move it again with another graphics statement. If you use a more complex command that uses two addresses, like a command that draws a line or a box, the graphics cursor starts at the first address and ends up at the second one.

There are two methods of positioning the graphics cursor on a pixel in a window in Amiga BASIC: absolute addressing and relative addressing.

ABSOLUTE ADDRESSING

Absolute addressing is the same addressing system you use to specify the opposing corners of a window in a screen (described in the WINDOW statement section in the previous chapter), except that it specifies an address within the boundaries of a window instead of a screen. In absolute addressing, the address you specify is always in reference to the upper left corner of the window.

The pixels are numbered by column and row, starting with (0,0) in the upper left corner of the window. The addresses are enclosed in parentheses and use a comma to separate the column number from the row number. For example, (13,20) specifies an address 14 columns to the right and 21 rows down from the upper left corner (0,0) of the window.

RELATIVE ADDRESSING

Relative addressing lets you describe a new pixel location as the distance from a previous pixel address. Each relative address starts with the keyword `STEP`, followed by a distance in columns and a distance in rows, separated by a comma and enclosed in parentheses. For example, `STEP(5,8)` specifies a new address five columns to the right and eight rows down from the last pixel address you used. If you want to use a relative address to move to the left or up from the last pixel address, you use negative numbers. For example, `STEP(-7, -9)` specifies a new address that is 7 columns to the left and 9 rows above the old address.

Relative addressing saves you the time you'd take to calculate a new absolute address. For example, if you use absolute addressing and you start with a pixel at location (10,5), then want to specify a pixel 10 columns to the right and 20 rows down, you have to add 10 to the column number and 20 to the row number to get a new address of (20,25). If you use relative addressing, you can specify the second address using `STEP(10,20)` to move 10 columns to the right and 20 rows down. Figure 5-1 shows how this works.

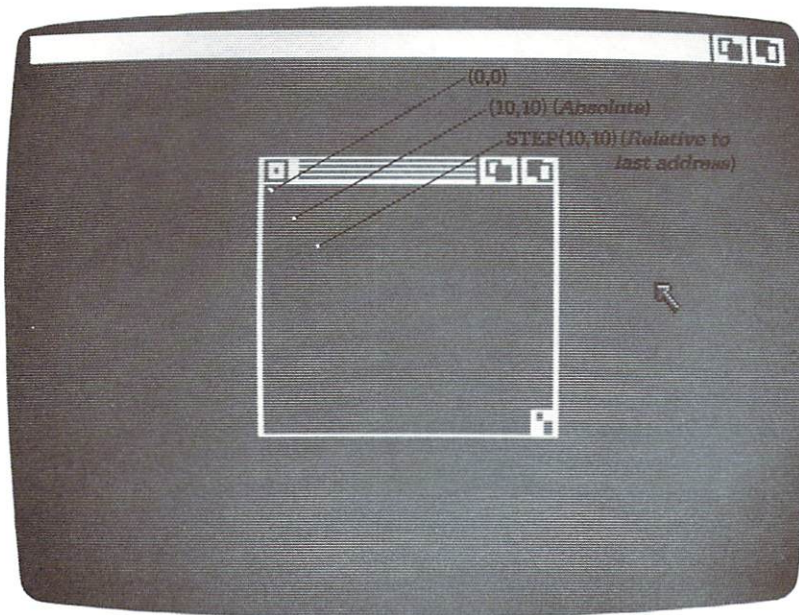


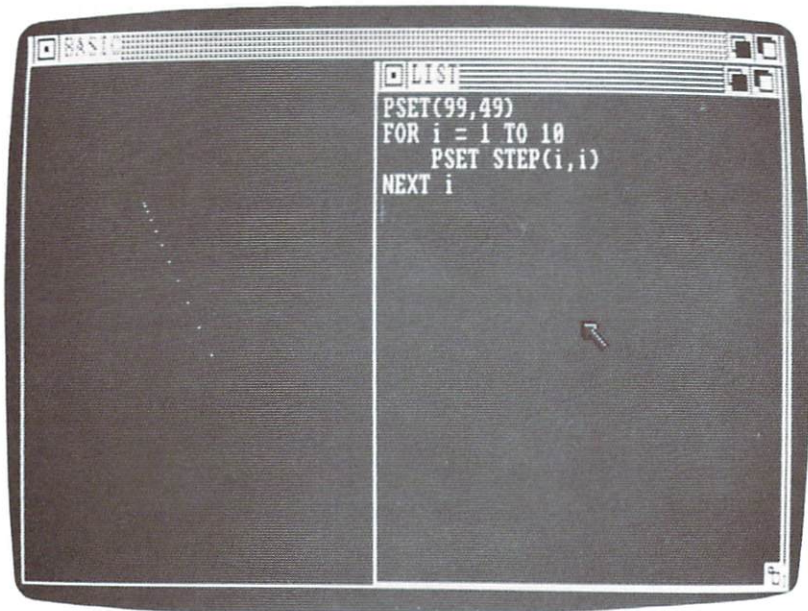
Figure 5-1.

Using relative addressing to specify a new pixel address.

Relative addressing works especially well in FOR...NEXT loops where the address changes in each cycle of the loop. Instead of calculating new absolute addresses for each cycle, a simple relative address will change the address by the incremental value of the loop for each iteration. For example, the program shown in Figure 5-2 uses the PSET statement (discussed shortly) to plot a point on the screen 100 pixels over and 50 pixels down using an absolute address, then uses another PSET statement in a FOR...NEXT loop to plot 10 more points, each one plotted in relation to the previous point specified.

Figure 5-2.

Points plotted using a FOR...NEXT loop and relative addressing.



Anytime you use a relative address, it uses the last position of the graphics cursor as its point of reference. That position is normally set by the last graphics command. Consider an example: If you use a line-drawing command that ends up at address (55,90), and then use a new graphics statement that uses the relative address STEP(10,10), BASIC uses (55,90) as its point of reference to compute a new address 10 columns to the right and 10 rows down. The key to controlling the graphics cursor with relative addressing is knowing its location before you use a relative address. If you don't, you may get unpredictable results. When you're not sure, it's a good idea to use absolute addressing.

WINDOW BOUNDARIES

When using pixel addresses in the drawing statements, it's important to make sure you remain inside the boundaries of the output window. Some statements simply won't work if you specify an address outside the window, while others will stop the program and return an error message.

When you first create a window with the WINDOW command, you set the size of the window's interior with opposite corner addresses. (See Chapter 4 for details on creating a window.) If you create a window with opposite corners at (50,20) and (250,120), with a title bar, no sizing gadget, and no front and back gadgets, you have a window interior that measures 201 pixels across (the distance from column 50 to column 250, inclusive) and 101 pixels from top to bottom (the distance from row 20 to row 120, inclusive). The window that appears on the screen is actually larger than 201 by 101 because it includes the border and the title, which are outside the interior area. It also includes an interior border that measures one pixel across the top and bottom, and two pixels wide on either side, just inside the window border, as you can see in Figure 5-3. Since the interior border is the same color as the background color, you can't really see it there. It's impossible to fill those pixels in, since this "invisible" interior border is always in place—you can't turn it off or disable it.

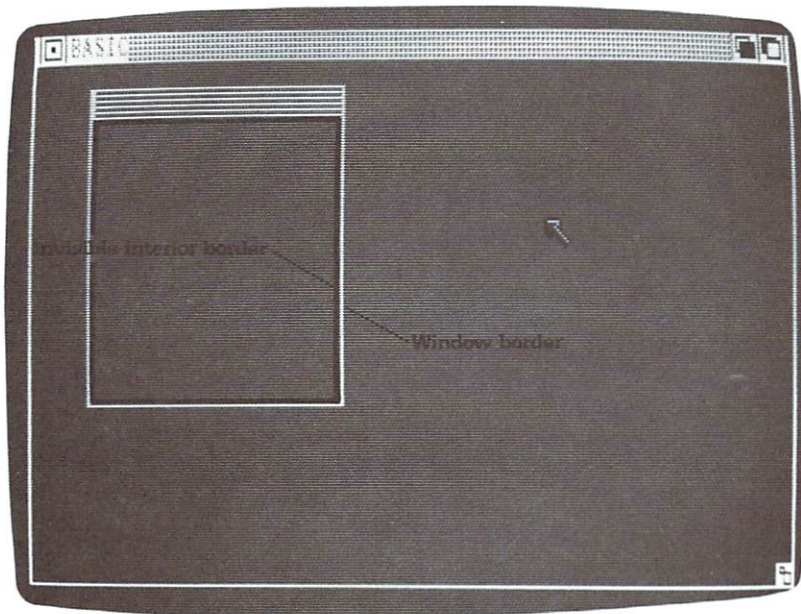


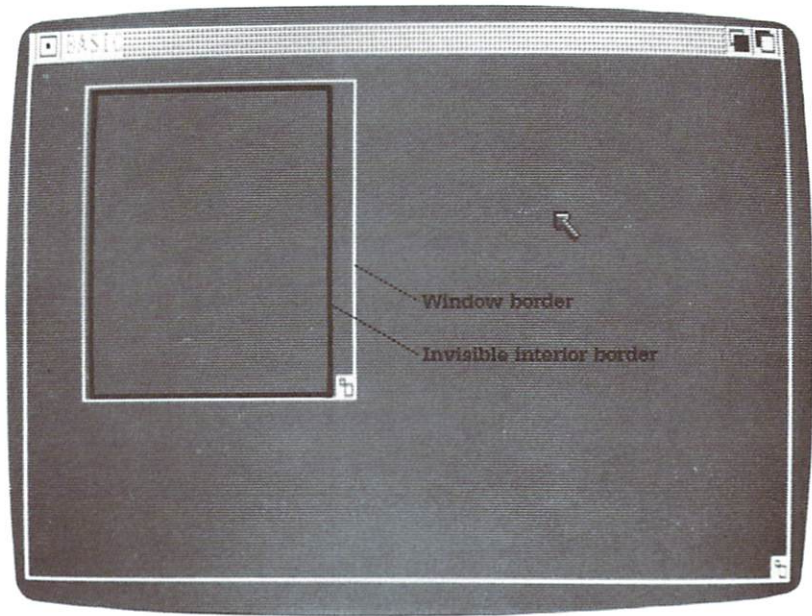
Figure 5-3.

The graphics area inside a window with opposing corners at (50,20) and (250,120).

If you add a sizing gadget to a window, the window becomes 15 pixels wider to accommodate the sizing gadget. You can't draw in the 10 columns next to the right border, though, so you gain only 5 columns of usable space in the window's interior. If you create a window without a title bar, the interior of the window moves up to occupy the space where the title bar would be, and you gain 10 more rows of pixels at the bottom of the window. Figure 5-4 shows a window with the same opposing corner addresses as the window in Figure 5-3, but with a sizing gadget and no title bar. You can see that it has an interior that measures 206 columns by 111 rows—5 pixels wider and 10 pixels higher than the same window with a title bar and no sizing gadget.

Figure 5-4.

The graphics area inside a window with opposing corners at (50,20) and (250,120), no title bar, and a sizing gadget.



When you use graphics statements in a window, you know your addresses will remain within the window if you keep them within the limits set by the opposing corner addresses and within the invisible border. Another way to keep them within window boundaries is to read the size of the window boundaries with the `WINDOW()` function, discussed in the next chapter.

THE CLS STATEMENT

Before you start creating figures in the output window with BASIC statements, or whenever you want to erase any existing figures in the output window, you can completely clear it by using the CLS command. Its format is simple:

```
CLS
```

CLS erases everything in the output window and fills it entirely with the background color. The following program demonstrates its effect. It uses the COLOR statement to change the background color four different times, then uses the CLS statement to fill the screen with each background color.

```
FOR i = 1 TO 3
  COLOR , i
  CLS
  FOR t = 1 TO 2000: NEXT t
NEXT i
COLOR , 0
CLS
```

THE PSET STATEMENT

PSET is a very simple statement: It changes the color of a single pixel in the output window to a color you specify. By combining many PSET statements, you can create intricate images in a window. The format is simple:

```
PSET(pixel address), color-register number
```

The pixel address can be either an absolute or relative address. The color-register number is optional. If you include it, it must be separated from the address by a comma. If you don't specify a color-register number, PSET colors the pixel with the foreground color. For example, the statement

```
PSET(149,99)
```

fills in the pixel at column 150, row 100, using the current foreground color. If you specify a color-register number, PSET uses the color

in that register to color the pixel instead of using the foreground color. For example,

```
PSET(199,149),3
```

uses color register 3 to fill in the pixel at column 200, row 150 instead of using the foreground color.

You can't specify a color register not used by the screen you're working with. For example, if you're using a 2-bit-plane screen, such as the default Workbench screen, you can only specify registers 0 through 3, since a 2-bit-plane screen allows only four color registers. Also, specifying a color-register number affects that pixel only—subsequent pixels set with the PSET statement will use the default foreground color unless another register is specified.

THE PRESET STATEMENT

The PRESET statement works almost exactly like the PSET command. The difference is that PRESET colors the pixel at the address with the background color instead of the foreground color if you don't specify a color register. It uses the same format as PSET:

```
PRESET(pixel address), color-register number
```

The color-register number is optional. For example, the command

```
PRESET(14,74)
```

fills in pixel (15,75) with the current background color. PRESET is very handy for erasing pixels previously created with PSET.

If you follow the pixel address with a comma and a color-register number, PRESET colors the pixel with the color in the specified color register (just like a PSET command that specifies a color register). Like the PSET statement, the number of color registers available to you depends on the number of bit planes defined for the screen containing the output window.

USING PSET AND PRESET WITH FORMULAS TO CREATE FIGURES

If you have the patience of a saint, the mathematical abilities of an accountant, and the design methods of a carpet maker, you can individually plot each pixel of an intricate image, using the PSET

statement with a different address and color for each pixel. If you want to do this, it helps to use graph paper to draw your figure and work out the pixel addresses and colors.

An easier way to create figures is to let a FOR...NEXT loop do some of the work for you. As a simple example, the following program draws a diagonal line on the screen:

```
FOR i = 1 TO 100
  PSET(i,i)
NEXT i
```

If you have some experience with trigonometry, you can use its functions to plot some nice shapes in the output window. By using the value of the counter variable in the FOR...NEXT loop as one of the addresses (either the column or the row number), and a trigonometric function to set the value for the other half of the address, you can actually chart values in the output window. As an example, the following program uses the SIN function to create a sine wave on the screen, as shown in Figure 5-5.

```
FOR i = 0 TO 600
  PSET(i, 90 + 80 * SIN(6.28318 / 600 * i))
NEXT i
Loop: GOTO Loop
```

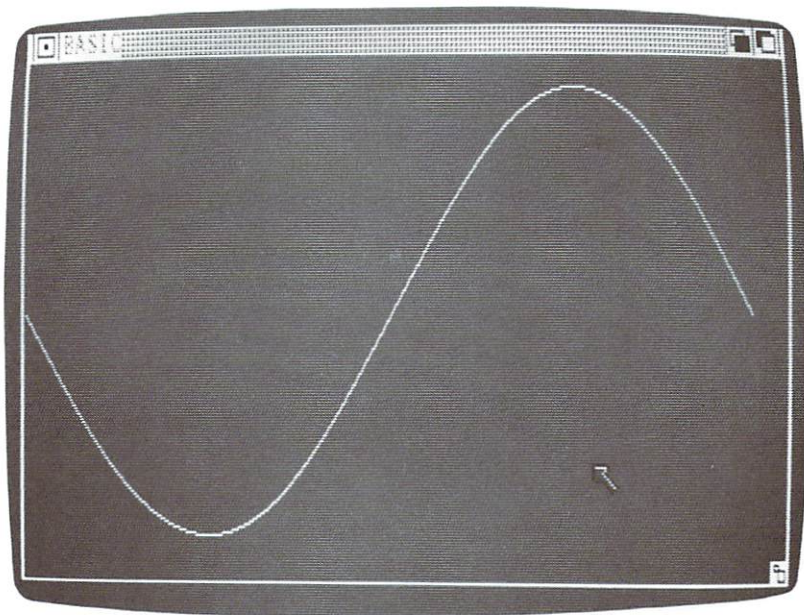


Figure 5-5.

The PSET command used with a trigonometric function creates this sine wave.

If you analyze the row address in the PSET statement, you can see that it uses one full cycle of a sine wave (6.28318, which is two times π), that it multiplies the results by 80 so the wave goes to the top and bottom of the window, then adds this value to 90 to center the wave horizontally in the window. The last line in the program is merely an "infinite loop" that keeps the program running and the List window hidden; otherwise, it might appear and cover up the right half of the wave. To end the program and show the List window, use the Menu button on the mouse and choose Stop from the Run menu.

Figuring out mathematical solutions to create figures may not be your cup of tea, but if you know how to use them, they can create some fascinating shapes, save you a lot of tedious single-pixel plotting, and give you a practical application for all that trig you slogged through in high school.

THE LINE STATEMENT

So far, you've read about Amiga BASIC statements that work with single pixels. BASIC also provides several statements that let you draw fully formed objects with just a single statement. One such statement is LINE. Actually, variations of the LINE statement let you draw three types of objects: straight lines, hollow boxes, or boxes filled with the foreground color. The LINE statement uses this format:

```
LINE(starting address) - (ending address), color-register  
number, box options
```

LINE has to have both a starting and an ending address, separated by a dash. The addresses can be either absolute or relative (relative addresses must be preceded by the keyword STEP). The color-register number and the box options are optional—you can leave them out. However, if you do include them, they have to be separated by commas.

CREATING LINES

All you need do to create a line is supply the beginning- and ending-point addresses, like this:

```
LINE(70,54) - (10,5)
```


As you can see in Figure 5-6, this statement draws a line from the point at (70,54) to the point at (10,5), using the default foreground color. To use any other color register for the line, just follow the statement with a color-register number:

```
LINE(70,54) - (10,5), 2
```

This statement draws the same line as before, but colors it using the color in color register 2 instead of the foreground color. You should note that specifying a color-register number affects only that line; subsequent lines drawn with no color-register number specified will use the foreground color. Also, the number of color registers available depends on the depth (number of bit planes) of the screen containing the output window.

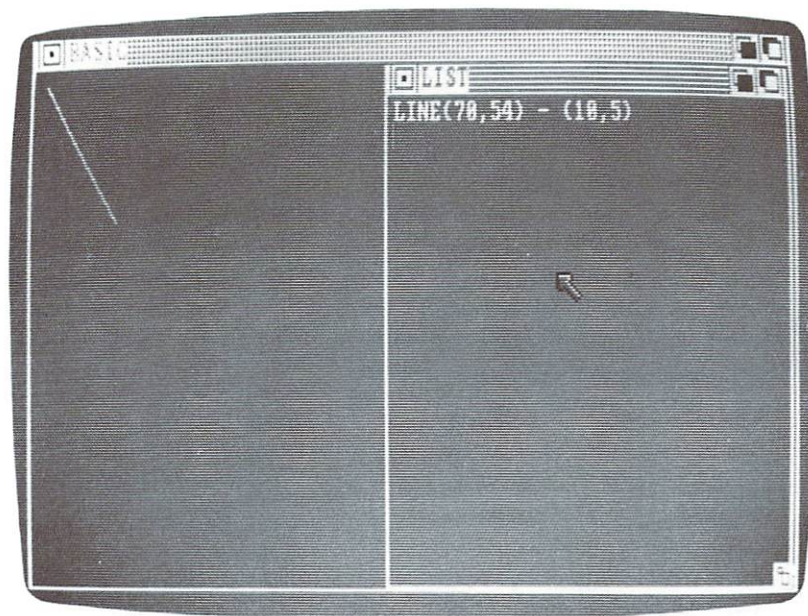


Figure 5-6.

The LINE statement creates a line.

CREATING BOXES

To create a box with the LINE statement, you specify the upper left and lower right corners of the box, and add a box option after the color-register number (if you want to use the current foreground color, you must include the comma at its position). There are two box options: B and BF. The B option will make LINE draw a hollow box instead of a line, using the starting and ending addresses as

opposing corners of the box. The BF option draws a box filled with the current foreground color. For example,

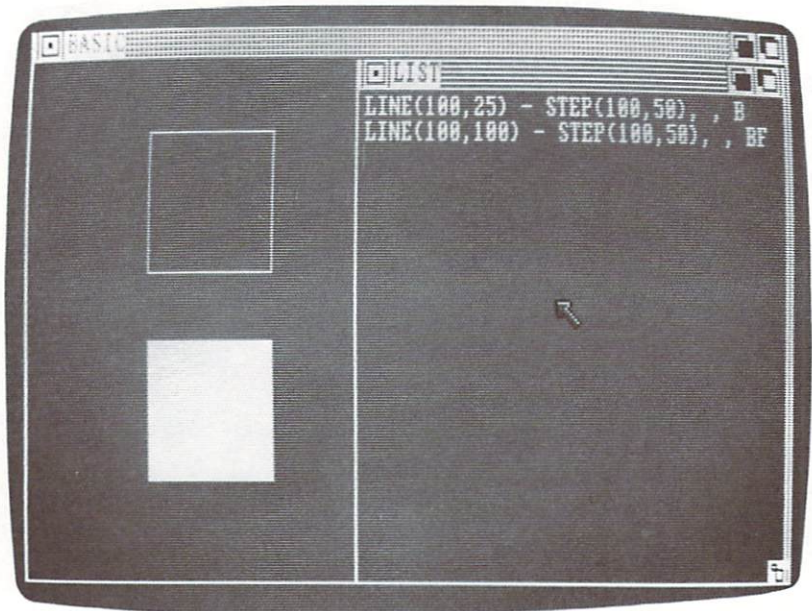
```
LINE(100,25) - STEP(100,50), , B
```

draws a box with its upper left corner at address (100,25) and its lower right corner 100 columns to the right and 50 rows down (recall that STEP indicates a relative address). In this example, the box is a hollow box, since we used the B option, and it's drawn with the current foreground color since no color register was specified. You can see the results in Figure 5-7, together with a filled box created with this statement:

```
LINE(100,100) - STEP(100,50), , BF
```

Figure 5-7.

The LINE statement with the B option creates a hollow box, and the BF option creates a box filled with the current foreground color.



THE CIRCLE STATEMENT

Just as variations of the LINE statement allow you to create straight lines, angles, and boxes, the CIRCLE statement lets you create circles, ovals, and arcs. CIRCLE uses this format:

```
CIRCLE(center address), radius, color-register number,  
arc starting point, arc ending point, aspect
```

You must specify the center address and radius values. The center address can be absolute or relative. The color-register number, the arc starting and ending points, and the aspect are all optional. All the values have to be separated with commas if you include them.

CREATING CIRCLES

To create a simple circle, you supply CIRCLE with an address for the center of the circle and a value for the radius. The radius is measured in pixels from the center of the circle to the edge of the circle. For example, the statement to create a circle centered at (150,100) with a radius of 50 pixels is:

```
CIRCLE(150,100), 50
```

You can see the result in Figure 5-8.

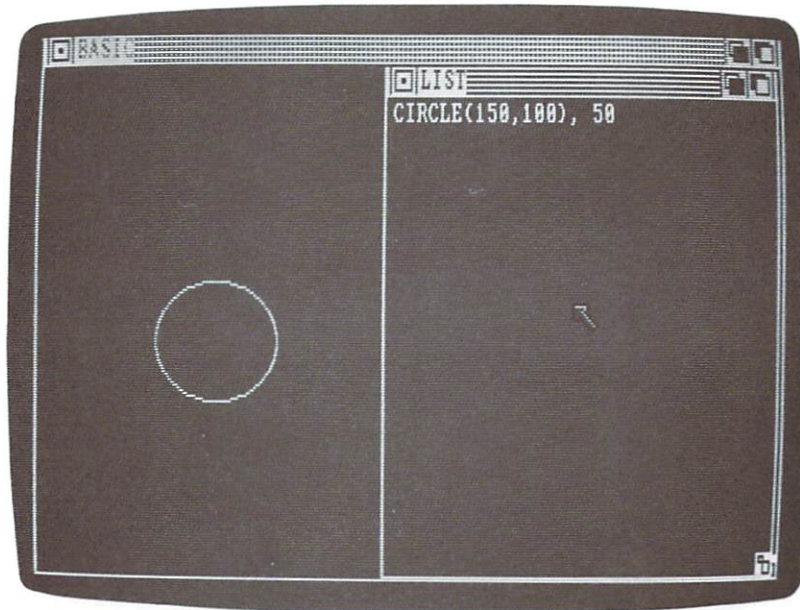


Figure 5-8.

The CIRCLE statement creates a circle.

CIRCLE normally creates a circle using the current foreground color. If you want to use a different color, follow the radius with a comma and a color-register number. The statement

```
CIRCLE(150,100), 50, 3
```

creates the same circle as before, but draws it using the color in color register 3.

CREATING ARCS

To create an arc, add a starting and ending point on the circle for each end of the arc. To specify these points, measure the circle in *radians* and give CIRCLE the two ends of the arc as radian measurements. Think of a radian as a unit of measurement that describes the diameter of a circle. A full circle is 6.28318 radians; a half circle is 3.14159 radians (mathematically referred to as π).

Radian measurement for the CIRCLE statement begins at the three o'clock position of the circle, where the measurement is zero. You measure from that point *counterclockwise* around the circle—twelve o'clock is 1.57079 radians (π divided by two), nine o'clock is 3.14159 (π), six o'clock is 4.71238 (π times 1.5), and a full circle is 6.28318 radians (two times π).

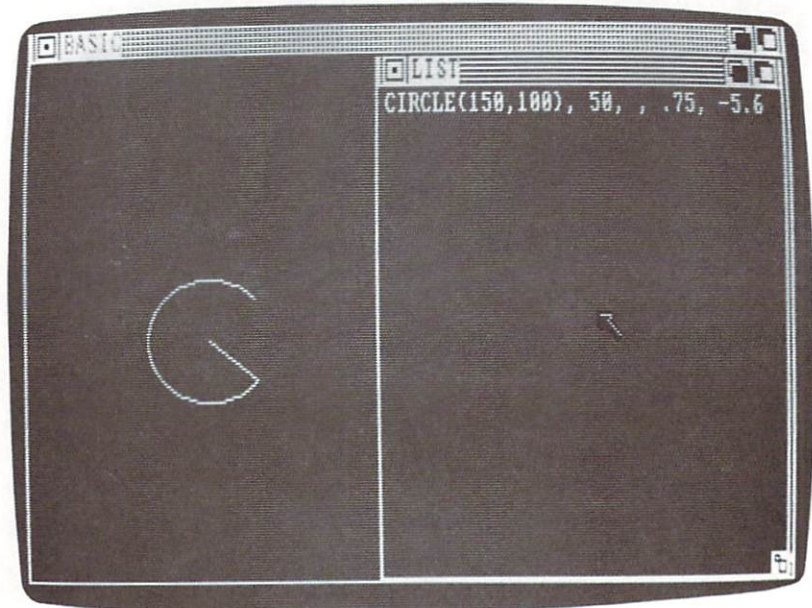
If the value specified for the starting point is smaller than the value specified for the ending point, the arc will be drawn counterclockwise. If the first value is larger than the second, the arc will be drawn clockwise.

In addition to drawing the arc itself, you can add straight lines from the starting and/or ending points of the arc to the center of the circle by making either or both the arc starting- or ending-point value a negative number. BASIC draws the arc as if the values were positive, then draws a line from the point(s) specified as negative numbers to the circle's center. This statement draws an arc with a straight line from its ending point, as shown in Figure 5-9:

```
CIRCLE(150,100), 50, , .75, -5.6
```

Figure 5-9.

The CIRCLE statement can create arcs and add lines from the starting and/or ending points to the center of the circle.



Many people are more familiar with measuring circles as degrees of arc rather than radians. To convert degrees into radians, you can use the following formula:

$$(3.14159 * (\text{angle of degree})) / 180 = \text{radians}$$

where *angle of degree* is the value of the degree you wish to convert. For example, an angle of 45 degrees would return the value of .785 radians.

You can see a circle measured in radians in Figure 5-10.

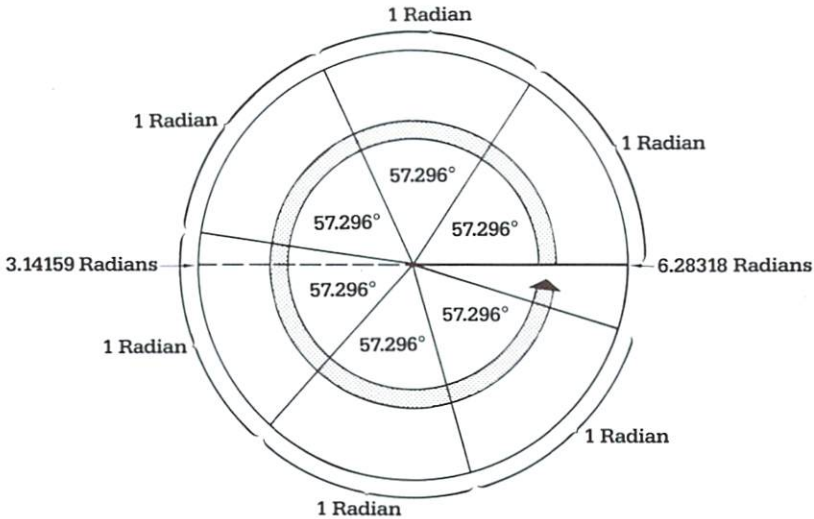


Figure 5-10.

A circle measured in radians.

CREATING OVALS

By changing the aspect of the CIRCLE statement, you can change the height of the circle it creates and turn it into an oval. The aspect follows the starting and ending points of an arc in the CIRCLE statement.

The formula that CIRCLE uses to create a circle assumes that pixels on the monitor screen are perfectly square. However, the monitor doesn't create absolutely square pixels, even in resolution modes 1 and 4—the pixels are slightly taller than they are wide. To compensate for this difference, Amiga BASIC normally uses a default aspect value that keeps circles looking perfectly round on the Amiga monitor.

If you don't specify an aspect, CIRCLE uses a .44 aspect automatically, a value that creates perfect circles in the default mode 2 screen. If you want to increase or decrease the pixel aspect ratio, just specify a new aspect value. If you aren't creating an arc, you

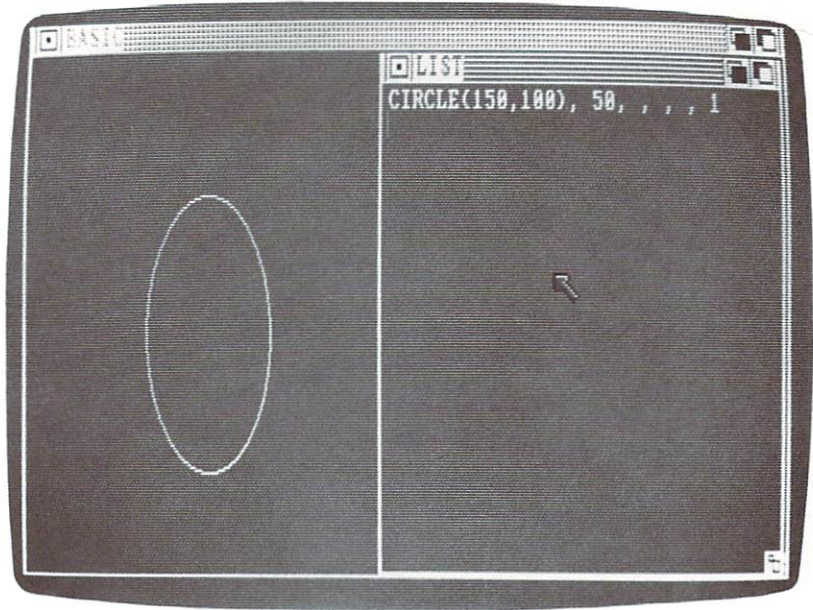
don't have to specify starting and ending points, but you do have to use commas to hold their places. As an example, increase the aspect value to stretch the circle you've been working with:

```
CIRCLE(150,100), 50, , , , 1
```

You can see the results in Figure 5-11.

Figure 5-11.

The CIRCLE statement creates an oval.



When you increase the aspect value above the default value, the circle is stretched out vertically; when you decrease the aspect below the default value, the circle is stretched out horizontally.

Since each screen-resolution mode uses a different size of pixel, you need to use different aspects to keep the circles perfect for each mode. In a mode 2 screen, the default value that Amiga BASIC uses will create round circles automatically—you don't have to specify an aspect value. Mode 1, mode 3, and mode 4 screens, however, require you to enter a value in order to get round circles. The following table gives you the circular aspect for each mode:

Screen mode	Aspect value
1	.88
2	.44
3	1.76
4	.88

If you set the aspect value to create ovals and also include starting and ending points for an arc, CIRCLE will create arcs that are curved just like the oval—just as the aspect normally would create if it were a complete circle.

CREATING MULTI-SIDED FIGURES

AREA and AREAFILL are two Amiga BASIC statements you use together to create multi-sided figures and fill them with solid colors or intricate patterns. They work a little like a connect-the-dots picture—first you set the invisible dots in the output window with AREA, and then use AREAFILL to connect the dots, and to fill the area with a color or pattern.

THE AREA STATEMENT

The AREA statement uses this format:

```
AREA (address)
```

The address can be either absolute or relative.

AREA is simple to use—just enter one AREA statement for each point you want to set in the output window. As an example, the statement

```
AREA (120,95)
```

sets an invisible point at address (120,95).

To create complex shapes, you use a series of AREA statements to set the points of the sides of the shapes. You can't set more than 20 points in a row without using an AREAFILL statement to connect them all together. Once you connect those points with AREAFILL, though, you can continue setting points for the same shape, or set points for a new shape.

It's often easier to use relative addresses with the AREA command after you set the first point with an absolute address. In the example below, which sets the points for a diamond shape, the first address is an absolute address and the next three addresses are relative addresses:

```
AREA (150,50)  
AREA STEP (100,50)  
AREA STEP (-100,50)  
AREA STEP (-100,-50)
```

THE AREAFILL STATEMENT

AREA statements by themselves have no effect on the appearance of the output window—they only set the points for the shape you're drawing. AREAFILL connects the points with lines and fills in the interior of the shape with either the foreground color by default, or with a color or pattern you specify. (If you want to create a shape that isn't filled in, you have to draw each line of the shape individually, using graphics statements other than AREA/AREAFILL.) AREAFILL uses this format:

`AREAFILL mode number`

The mode number is an option that specifies how to fill the area; you can omit it if you just want to use the foreground color.

In order for AREAFILL to work, you have to precede it with at least two AREA statements. AREAFILL first connects the points you set with the AREA statements in the order that you set the points. When it reaches the last point you set, it connects it to the first point you set, so you don't have to end your series of AREA statements with the address that you started with (although it won't hurt if you do).

After the points are connected, AREAFILL fills the interior and the boundaries of the shape with the foreground color if you haven't specified a mode number. Since it fills in the boundaries as well as the interior, the shape is one solid color; it doesn't have an outline in a different color.

You can specify a mode number if you want to control the color of the shape. A 0 will color the shape with a pattern you've created with the PATTERN statement (discussed later) or, if you haven't defined a pattern, with the foreground color. If you specify a mode number of 1, AREAFILL changes the colors of all pixels within the area defined by the AREA statements to their inverse colors. A color's inverse color is found in the color register at the opposite end of the palette. For example, the table below shows the inverse colors of an eight-color (3-bit-plane) screen:

Color register	Inverse-color register
0	7
1	6
2	5
3	4
4	3
5	2
6	1
7	0

If you omit the mode number, AREAFILL assumes a 0 and uses the current pattern (or the foreground color, if no pattern has been defined).

To see AREAFILL do its work, add it to the four AREA statements you used before to set the points of a diamond shape and then run the program:

```
AREA(150,50)
AREA STEP(100,50)
AREA STEP(-100,50)
AREA STEP(-100,-50)
AREAFILL
```

You can see the diamond shape appear on the screen, filled with the foreground color, in Figure 5-12.

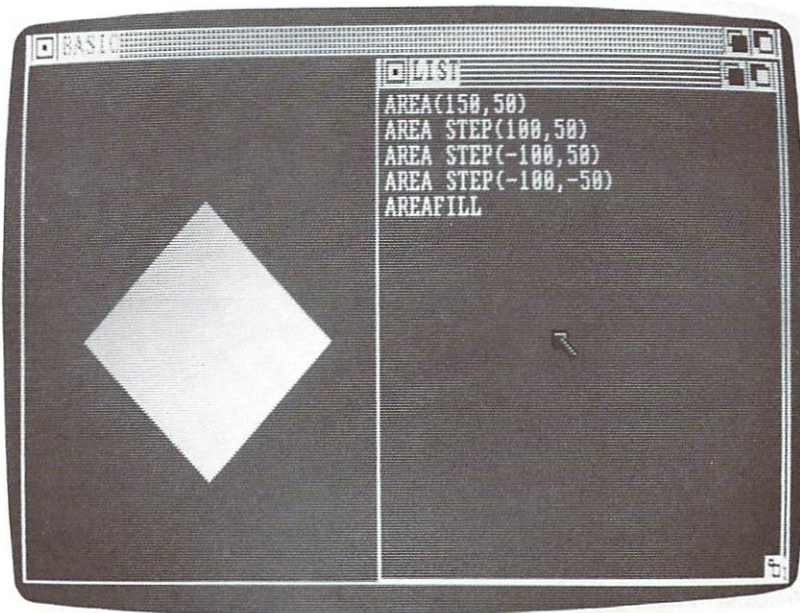


Figure 5-12.

The AREA and AREAFILL statements create a diamond shape.

If you change the last line of the program to

```
AREAFILL 1
```

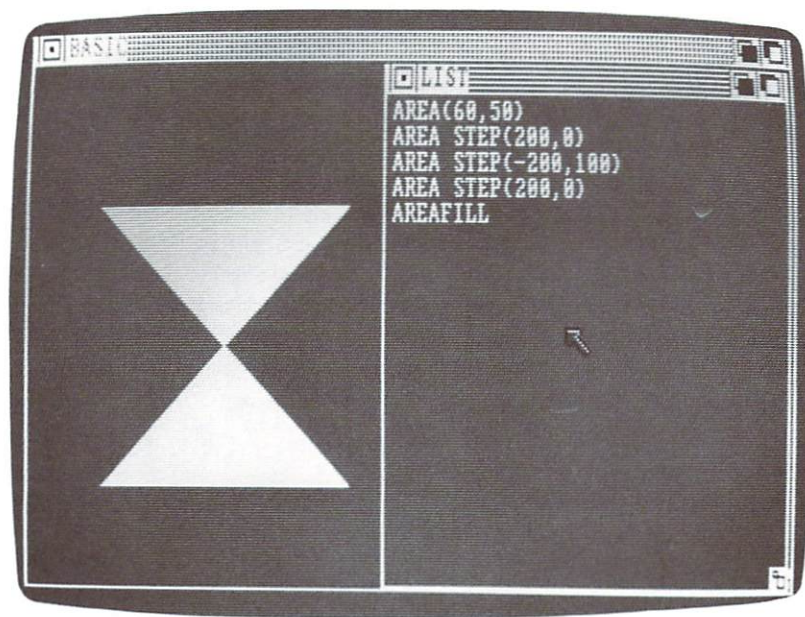
and run the program again, you'll see the same diamond shape repeated on the screen, but instead of filling it with the foreground color, the diamond will appear in the inverse color of the background pixels. Since there are no pixels of any other color within the AREA boundaries, AREAFILL 1 inverts just the color of the background pixels.

As you create shapes with AREA commands followed by AREA FILL, you don't have to keep the boundaries of the shapes from crossing each other. For example, this short program creates two filled triangles because the side boundaries cross each other, as you can see in Figure 5-13:

```
AREA(60,50)
AREA STEP(200,0)
AREA STEP(-200,100)
AREA STEP(200,0)
AREA FILL
```

Figure 5-13.

You can cross boundary lines with AREA statements.



FILLING A SHAPE WITH A PATTERN

If you create a pattern with the PATTERN command (described later in this chapter), AREA FILL will fill the shape with that pattern, combining the background and foreground colors, instead of just filling the shape in with the foreground color. If you set the AREA FILL mode to 1, the background and foreground colors in the pattern are displayed using color register 0 for the background color and the last color register (the inverted color register) for the foreground, with a one-pixel-wide border the color of the background around the shape.

THE PAINT STATEMENT

As you've seen, the LINE statement gives you the option to fill in the boxes you create with color. Similarly, you can use AREAFILL to fill in with a color or a pattern the shapes you create with AREA statements. Amiga BASIC also allows you to fill in any other enclosed shapes you've created (with the CIRCLE statement, or with a series of PSET or LINE statements, for example). Your tool for doing this is the PAINT statement. Its format is:

```
PAINT(address), fill color-register number, border color-  
register number
```

The address can be either absolute or relative. The register numbers for the paint and border colors are optional: If you include them, they must be separated by commas. If you omit the fill color-register number, PAINT assumes that the color you want to fill with is the foreground color. To fill with a different color, follow the address with a comma and a color-register number. PAINT will fill the shape with the color in that register.

An important point to keep in mind when using the PAINT statement is that it must be used in a refreshing window; that is, a window created using a window-features number of 16 or higher.

Before you use PAINT, you have to create the outline of a shape. You can use any of the graphics statements such as CIRCLE, LINE, or PSET to create the shape, but the outline has to be unbroken, and it has to be all in one color. If it isn't, PAINT will fill the interior of the shape and then escape through the break or second outline color to fill the entire screen.

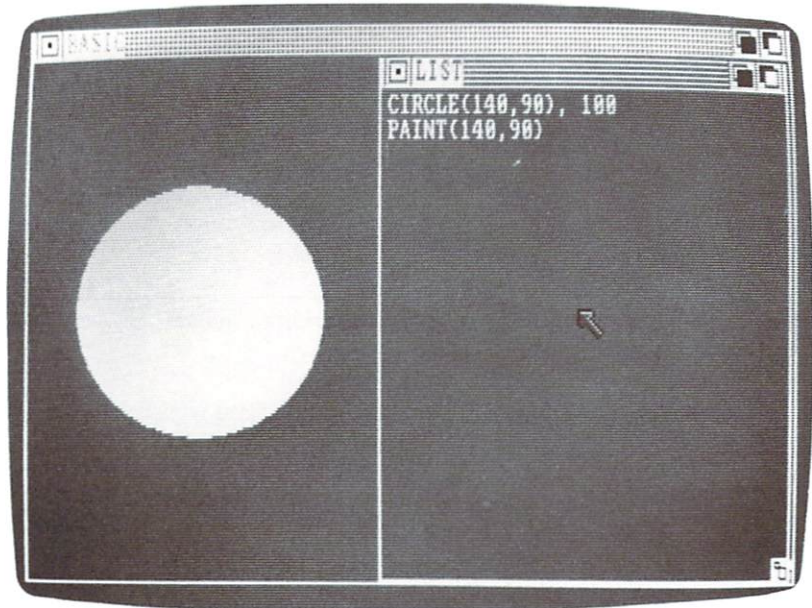
When you use PAINT, the address specifies the point in the output window where you want to start filling with color. This point can be inside or outside the outlined shape. PAINT will fill the screen around the point with the foreground color until it meets the outline of the shape or the boundary of the output window. If you start at a point outside the shape, PAINT changes the color of the screen all around the shape, leaving the interior its original color. If you start at a point inside the shape, PAINT fills in the shape, leaving the area around the shape its original color.

The following short program creates the circle displayed in Figure 5-14 (on the next page), and fills it with the current foreground color:

```
CIRCLE(140,90), 100  
PAINT(140,90)
```

Figure 5-14.

The PAINT command fills in a circle.



To fill a shape with a color other than the current foreground color, you just include the register number of the color you want. This PAINT command fills with the color in register number 2:

```
PAINT(140,90), 2
```

If you try this PAINT statement with the last circle you created, it will paint over the boundaries of the circle and fill the entire output window. That's because PAINT only recognizes boundaries of the same color as the color it uses to fill with. If you want to fill a shape with a color other than its border color, you can follow the fill color-register number with a comma and the color-register number of the color you want to use as a border. As an example, here's PAINT set up to fill with the color in register 3, recognizing the color in register 1 as a boundary:

```
PAINT(120,100), 3, 1
```

If you enter and run the following program, you'll see the circle in Figure 5-15, with an outside boundary of foreground color, filled inside with color 3:

```
CIRCLE(120,100), 100  
PAINT(120,100), 3, 1
```

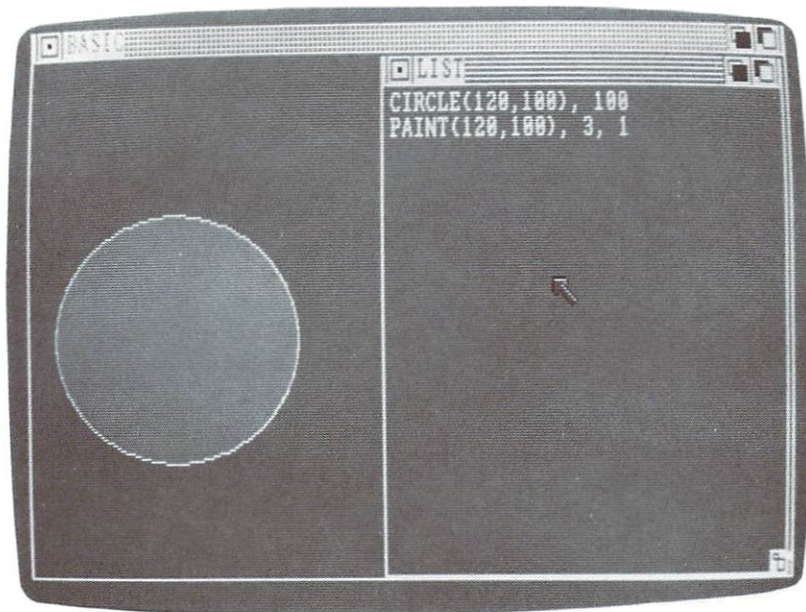



Figure 5-15.

A circle drawn in the foreground color, filled with the color in register number 3.

If you use the `PATTERN` statement (described next) to set up a pattern, `PAINT` will fill with that pattern instead of a solid color and will use the foreground color for the pattern. If you specify a fill color-register number, `PAINT` will still use the pattern to fill the shape, but will use the color in that register instead of the foreground color for the pattern.

THE PATTERN STATEMENT

When you use Amiga BASIC to draw lines and fill in shapes and areas, you usually draw and fill with solid lines and colors. If you want to add some variety and texture to your graphics creations, you can use the `PATTERN` statement to create patterns that you can use to draw lines and fill in areas. Once you've created and set these patterns, any drawing commands that follow a `PATTERN` statement will use the line and area patterns instead of solid lines and colors.

`PATTERN` uses this format:

```
PATTERN line mask, pattern-mask array name
```

You can omit either the line mask or the pattern-mask array name from the `PATTERN` command, but you can't omit both of them.

CREATING A LINE PATTERN

To create a line pattern, you must first create a 16-bit mask—that is, a binary (base 2) number that uses 16 ones and zeros. Each one in the mask stands for the foreground color, each zero stands for the background color. The binary number 1010101010101010, for example, is a mask that alternates the foreground and background colors every pixel. Once you set the line pattern, any lines you draw with Amiga BASIC will use the pattern in the mask (from left to right) over and over as many times as necessary—if a line is longer than 16 pixels, Amiga BASIC repeats the mask until the line is finished. If the line is shorter than 16 pixels, Amiga BASIC starts at the beginning (left end) of the mask and uses as much of the pattern as it can.

If you want to create a dashed-line pattern, try the line mask 1111000011110000, which alternates four pixels of foreground color with four pixels of background color. If you want a mostly solid line with small breaks of background, try 1111111111110000. You can experiment to find what suits you best when you use the PATTERN statement to put your line mask into effect.

Converting a binary line mask into a hexadecimal number

The PATTERN statement doesn't accept line masks as a binary number using ones and zeros, since it can't tell that the number is a binary (base 2) number. Amiga BASIC would interpret a 16-bit binary number as a very large number, in the quadrillions. You have to convert the binary bit mask into either a decimal (base 10) number or a hexadecimal (base 16) number, which is what the PATTERN statement expects to find. It's much easier to convert binary to hexadecimal, which PATTERN will recognize. To convert the bit mask into its hexadecimal equivalent, just break the bit mask into groups of four bits, and interpret each group using this table:

Binary	Hexadecimal	Binary	Hexadecimal
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

For example, the mask 1110011111100111 would break into 1110 0111 1110 0111. Using the table, these four groups translate into E, 7, E, and 7, so the entire hexadecimal number is E7E7. In Amiga BASIC, hexadecimal numbers are always preceded by &H, so the mask becomes &HE7E7 in hexadecimal. If the &H isn't specified, PATTERN assumes you are using a decimal (base 10) number. Although you can use a decimal number, you will probably find it easier to work with hexadecimal values.

Applying a line pattern in the PATTERN statement

Once you've designed a line pattern, turned it into a binary mask and then converted it into a hexadecimal number, you can put it into effect by applying it in a PATTERN statement. To draw a box using the line pattern you just created, try this short program:

```
PATTERN &HE7E7  
LINE(65,40) - STEP(150,100), , B
```

In Figure 5-16, you can see how Amiga BASIC draws the box using the line pattern you set with the &HE7E7 line mask. If you continue using the LINE statement, Amiga BASIC will use the same pattern to draw any other lines and boxes you create. The line pattern won't affect any circles or arcs drawn with the CIRCLE statement, though.

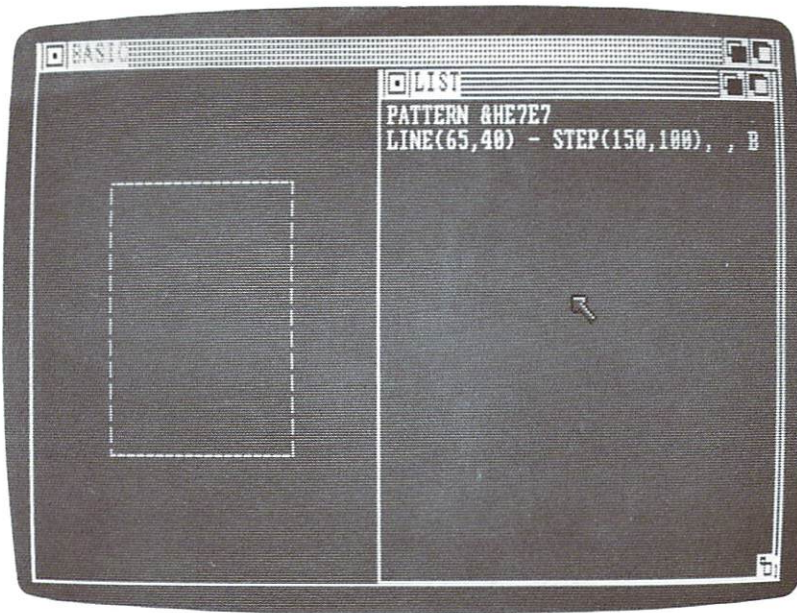


Figure 5-16.

A box drawn with dotted lines set by the PATTERN command.

To set the line pattern back to a solid line, use

```
PATTERN &HFFFF
```

which sets the line mask as a solid line (all ones).

CREATING AN AREA PATTERN

A line pattern is one-dimensional—you can set it by creating a single, 16-bit binary mask that is a string of ones and zeros. To create an area pattern, you have to create a two-dimensional mask using an array of ones and zeros. For an area, `PATTERN` uses an array that's always 16 bits wide, and two or a power of two high. In other words, a bit mask for an area is like a pile of 16-bit line masks—either 2, 4, 8, 16, or another power of two high. To create the mask, it's helpful to design it first on a piece of graph paper marked 16 squares wide, and as high as you want to make the area (as long as it's a power of two). Mark it in with ones where you want the foreground color to appear, and zeros where you want the background color. When you're finished, you should end up with a column of 16-bit binary numbers. As an example, this 8-line-high area mask creates a square polka-dot pattern:

```
0000111100001111
0000111100001111
0000000000000000
0000000000000000
1111000011110000
1111000011110000
0000000000000000
0000000000000000
```

Putting the area pattern mask into an integer array

To store the area pattern mask for the `PATTERN` statement, you have to put it into an integer array. To do so, you first have to translate each line into a hexadecimal number, just as you did for the line mask. The polka-dot area mask you just created translates as:

```
0F0F
0F0F
0000
0000
F0F0
F0F0
0000
0000
```

After you translate each line of the mask, you count the number of lines and create an integer array in your program using exactly the same number of elements as you have lines. For the polka-dot pattern in our example, you need to specify eight elements (one for each line), so you dimension the array to 7 with a DIM statement. (Since each array starts with element number 0, you actually have 8 elements in an array dimensioned to 7.) You then assign the lines of the mask in top-to-bottom order to the elements of the array. The program lines below assign the polka-dot mask to the integer array `POLKADOT%` (the `%` signifies an integer array):

```
DIM POLKADOT%(7)
POLKADOT%(0) = &H0F0F
POLKADOT%(1) = &H0F0F
POLKADOT%(2) = &H0000
POLKADOT%(3) = &H0000
POLKADOT%(4) = &HF0F0
POLKADOT%(5) = &HF0F0
POLKADOT%(6) = &H0000
POLKADOT%(7) = &H0000
```

Applying an area pattern in the PATTERN statement

Once you've created an area pattern, you apply it by putting the name of your integer array after the line-pattern mask in the `PATTERN` command, separating it with a comma. If you don't want to apply a line-pattern mask, you can leave its place empty, insert a comma, and then include the integer array name. Any Amiga BASIC graphics command you use afterward that fills in an area will use the graphics pattern you set. The following command sets the area pattern you just designed:

```
PATTERN, POLKADOT%
```

The following short program brings everything together to draw the box filled with polka-dots shown in Figure 5-17:

```
DIM POLKADOT%(7)
POLKADOT%(0) = &H0F0F
POLKADOT%(1) = &H0F0F
POLKADOT%(2) = &H0000
POLKADOT%(3) = &H0000
POLKADOT%(4) = &HF0F0
POLKADOT%(5) = &HF0F0
POLKADOT%(6) = &H0000
POLKADOT%(7) = &H0000
PATTERN, POLKADOT%
LINE(70,10) - STEP(150,160), , BF
```

Figure 5-17.

A polka-dot box
created with the LINE
and PATTERN
statements.



If you type this program yourself, you may notice that BASIC abbreviates the numbers you type in for each array element. For example, after you type &H0000 and press RETURN, BASIC converts the number to &H0, which has the same value.

To turn the area pattern back to a solid foreground color, you create a simple two-element integer array, assign a solid pattern mask, then apply it in a PATTERN command:

```
DIM SOLID%(1)
SOLID%(0) = &HFFFF
SOLID%(1) = &HFFFF
PATTERN, SOLID%
```

CHANGING PATTERN COLORS

Keep in mind as you use line and area patterns that you're not locked into any specific colors. The ones and zeros in the mask just specify which bits will use the foreground and background colors. You can use the COLOR statement to change the background and foreground colors to different registers or use the PALETTE statement to change the colors in the background and foreground color registers to create some vivid patterns.

In this chapter, you've learned Amiga BASIC statements that color pixels, draw lines, and create shapes filled with colors and patterns. You can use these statements together to create images on the screen. In the next chapter, you'll learn how to add text to your images, put the images in windows that change size, and copy part of your image and transfer it to another location in a window.



**CHAPTER SIX
AMIGA BASIC
GRAPHICS: ODDS
AND ENDS**

Amiga BASIC has a number of miscellaneous graphics statements and functions that don't create screens and windows and fill them with graphics. Instead, these "odds and ends" statements and functions allow you to label your graphics, change their size with the size of a window, or copy them from one area of the screen to another.

You can use several of these statements and functions to work creatively with text; you can add text to your pictures with the PRINT statement, and you can use the LOCATE and COLOR statements to position your text in just the right location with the colors you want. To keep track of the text's location, you can use the CSRLIN and POS(0) functions.

Other statements and functions help you adapt your graphics to the changing dimensions of a resizable window. You can use the WINDOW() function within the program to read the window's width and height, and use this information to change the graphics to fit the window's new size when the program user changes it with the sizing gadget. You can also use the WINDOW() function to check the window's ID number, to see how many colors it will support, and to examine other pertinent window information. To check the color of an individual pixel within the window, you can use the POINT function.

The GET and PUT statements allow you to cut and paste graphics in a window. With GET and PUT, you can also present a series of pictures in rapid succession in the output window.

CHARACTER ADDRESSING

When you print text characters in the output window to accompany your graphics, it's important to be able to position the characters accurately. In order to do so, you need to know how to use character addressing.

You use an entirely different addressing system to specify a text-character address than you use to specify a pixel address. Instead of counting columns and rows of pixels, you count lines and columns of text characters. Instead of moving an invisible graphics cursor from pixel to pixel, you move an invisible text cursor from character to character.

Text-character addresses start with a different pair of numbers than pixel addresses; instead of lines and columns starting at 0, text lines and columns start at 1. For example, the character in the upper left corner of the window is in address 1,1, not address (0,0). Text-address numbering starts in the upper left corner of the window, and the numbers increase as you move to the right or move down the screen, just like the numbering system for pixel addresses. Figure 6-1 shows how the lines and columns of text are numbered.

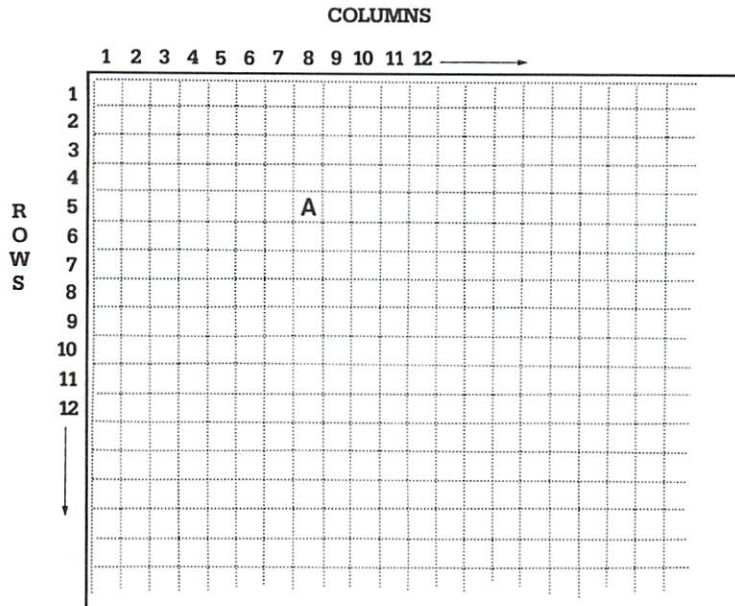


Figure 6-1.

Line and column numbering for text addresses.

Instead of specifying first the column and then the row as a pixel address does, a text-character address specifies first the line and then the column. You separate the two numbers with a comma, but you don't enclose them in parentheses as you do a pixel address. For example, a character in the fourth line down the screen seven columns to the right is at address 4,7.

USING DIFFERENT FONT SIZES

The size of the characters in the font you use with Amiga BASIC determines the number of text lines and columns that fit on the screen, so of course this affects the numbers you use in a text address. When you first open Amiga BASIC, it uses the same font

size you're working with in the Workbench. The Preferences program on the Workbench disk allows you to change the font you use on the Workbench screen.

There are two font-size options available: the default 60-column font and the 80-column font. The 60-column font uses characters that measure 10 pixels wide by 9 pixels high, wide enough to show up clearly on TV sets and composite video monitors. The 80-column font uses smaller characters to fit more text on the monitor screen. These characters measure 8 pixels wide by 8 pixels high, skinnier and shorter than 60-column characters. 80-column characters aren't always legible on TV sets and composite video screens, but they're very readable on RGB monitors.

The size of the characters in a window also depends on the resolution of the screen the window belongs to. All characters in the font have the same size as measured in pixels, but since the size of the pixels changes with different screen resolutions, the overall size of the characters changes also. For example, characters are twice as wide in a screen that uses low-resolution pixels as they are in a screen that uses high-resolution pixels. Characters are twice as high in a screen that uses non-interlaced pixels as they are in a screen that uses interlaced pixels.

The combination of font and resolution determines how many characters will fit in a full-screen window. The following chart shows how many rows and columns of text fit in a full-screen window with a title bar and sizing gadget.

	Mode 1	Mode 2	Mode 3	Mode 4
60-column font:	20 rows 30 cols	20 rows 61 cols	42 rows 30 cols	42 rows 61 cols
80-column font:	23 rows 38 cols	23 rows 77 cols	48 rows 38 cols	48 rows 77 cols

In Figure 6-2, you can see screens of four different resolutions on the monitor at once. Each screen displays the same sentence using the same font. Notice the difference in character size.

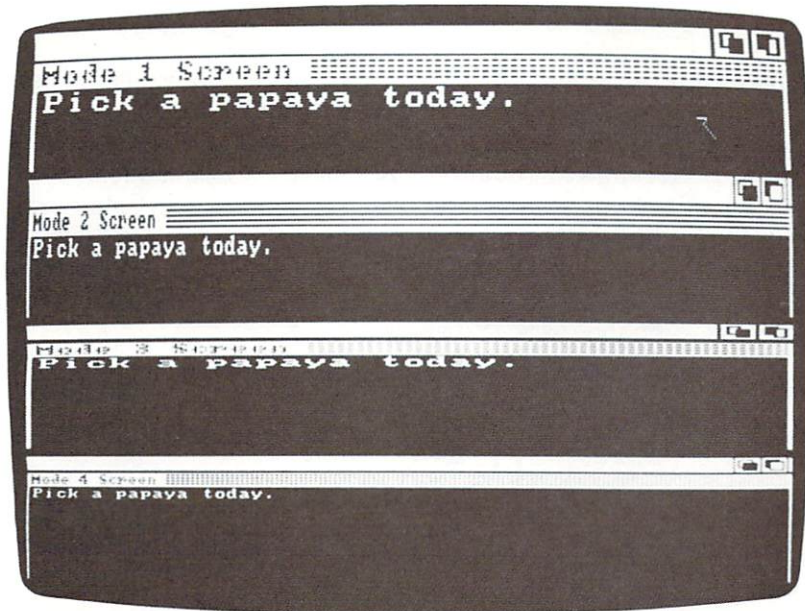


Figure 6-2.

The size of text characters depends on the resolution of the screen they're printed on.

THE PRINT STATEMENT

Every BASIC programmer is familiar with the PRINT statement—you use it to print messages on the screen. PRINT uses this format:

```
PRINT expression list
```

The expression list can be any combination of strings, constants, numeric variables, and string variables, separated with either a comma or a semicolon.

It's important to know where each PRINT statement leaves the text cursor when it's done printing. If the expression list ends in a comma, the text cursor stays on the same line but moves to the next comma stop, somewhat like a tab stop on a typewriter. If the expression list ends in a semicolon, the text cursor stays on the same line at the end of the material it just printed. If there is neither a comma nor a semicolon at the end of the expression list, the text cursor jumps to the beginning of the next line. You can get more precise information about text-cursor placement in the description of the PRINT statement in the Amiga BASIC manual.

THE LOCATE STATEMENT

If you mix text with graphics in a window and want to print text at an exact character address to fit with the graphics, trying to position the text cursor with numerous PRINT statements using commas and semicolons can be very tedious and sometimes impossible. Not only do you have to use a separate PRINT statement for each line you want to move down the screen, but if you go too far down the screen, the text might scroll up and ruin the position of everything else. For precisely positioning text, you're much better off using the LOCATE statement.

The LOCATE statement specifies a location on the screen where the next PRINT statement you use will start printing. It uses this format:

```
LOCATE line number, column number
```

The line and column numbers that specify the text-character address are optional, but if you don't supply them, the LOCATE statement won't relocate the text cursor. You can specify both the line number and the column number, or you can omit one or the other. If you leave out the line number, you need to put a comma before the column number. If you include only the line number, you don't need a comma at all.

The following LOCATE statements show examples of the three possible types of addresses:

Statement	Result
LOCATE 9, 24	Moves the text cursor to line 9, column 24.
LOCATE 4	Moves the text cursor to line 4 and uses the column number of the text cursor's current address.
LOCATE , 45	Moves the text cursor to column 45 and uses the row number of the text cursor's current address.

LOCATE STATEMENT EXAMPLES

When you use the LOCATE statement with the PRINT statement, you can position text accurately on the screen. In this short program, a FOR . . .NEXT loop changes the address in the LOCATE

statement in a series from 1,1 to 19,19 to print a diagonal line of words across the screen:

```
FOR i = 1 TO 19
  LOCATE i, i
  PRINT "Diagonal"
NEXT i
```

You can see the results in Figure 6-3.

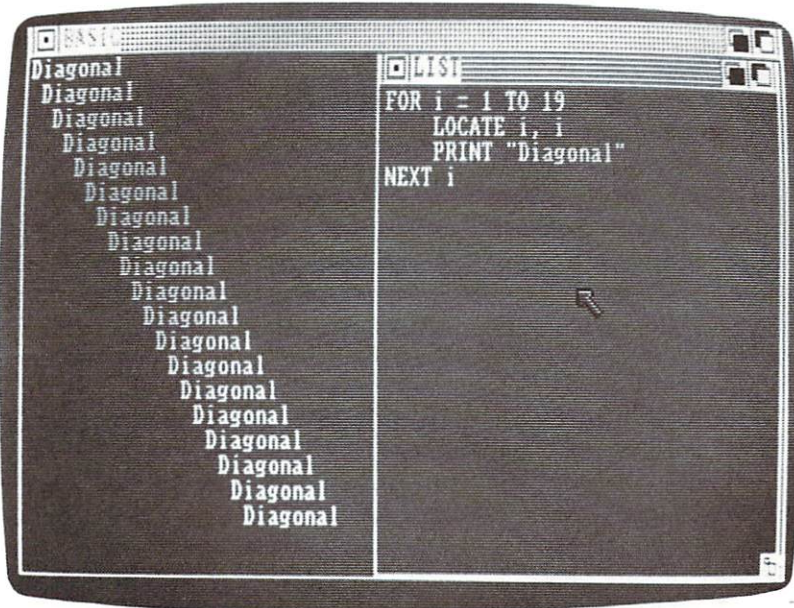


Figure 6-3.

LOCATE in a FOR ... NEXT loop creates a diagonal line of words.

Another good use of LOCATE is to keep the text cursor in the same location so that the PRINT statement following it prints new text that covers the old text. The following program counts from 1 to 100, printing the numbers in the same spot on the screen:

```
LOCATE 11, 5
PRINT "The count is now:";
FOR i = 1 TO 100
  LOCATE 11, 22
  PRINT i
NEXT i
```


LOCATING THE TEXT CURSOR WITH THE CSRLIN AND POS(0) FUNCTIONS

Many times you'll want to align some new text with text already on the screen. You can use the LOCATE statement, but it may be difficult to find the exact location you need. For example, to figure the address for the second LOCATE statement in the last program, you have to add up the characters in the string *The count is now:* and add the total to the location of the first LOCATE statement along with an extra count to add a space after *now:*. You can use two functions, CSRLIN and POS(0), to locate and record the address of the text cursor at any time for use in a later LOCATE statement.

CSRLIN returns the line number of the text cursor's position. You can assign the value it returns to a variable for storage. For example,

```
x = CSRLIN
```

stores the line number of the text cursor's current position in the variable x.

POS(0) returns the column number of the text cursor's position. Like the CSRLIN function, you can assign the value POS(0) returns to a variable for storage. For example,

```
y = POS(0)
```

stores the column number of the text cursor's current position in the variable y.

By using both functions at once, you can record the current address of the text cursor at any time using two variables. For example, the following program is a revised version of the last counting program. Its results are the same, but the second LOCATE statement doesn't use a specific address—it uses the text-cursor location taken at the end of the PRINT statement, stored in x and y, to set a new address one column past the end of *The count is now:*.

```
LOCATE 11, 5
PRINT "The count is now:";
x = CSRLIN
y = POS(0)
FOR i = 1 TO 100
    LOCATE x, y
    PRINT i
NEXT i
```

Notice the semicolon at the end of the first PRINT statement; it keeps the text cursor at the end of the string it just printed, so

CSRLIN and POS(0) can read its address. Notice also that the second LOCATE statement put the text cursor directly after the end of the *The count is now:*, but the number is printed with a space before it. That's because the PRINT statement always inserts a space before a positive number. To position text effectively, it's important that you know where the PRINT statement leaves the text cursor.

USING THE COLOR STATEMENT WITH PRINT

In the last chapter, you used the COLOR statement to set the foreground and background colors for the drawing statements. You can also use COLOR to set the foreground and background colors for text. When PRINT puts characters on the screen, it uses the foreground color to create the characters and the background color to create the background for the characters.

If you try the following statements, you'll see text printed with color 3 on a background strip of color 2:

```
COLOR 3, 2
PRINT "New colors!"
```

If you want to get back to the default text colors, use:

```
COLOR 1, 0
```

If you're putting text on top of graphics figures, you can set the background color of the text to match the figure's colors and blend the characters into the figure, or you can set the background color to a different color to make the text stand out as a strip against the figure. The following program draws three shapes in different colors in the output window and then uses LOCATE to print labels on each shape. The background and foreground colors of the text are changed for each label—in two of the shapes, the text's background matches the shape's color. In the third shape, the text's background is set to a different color to make the text stand out. The LOCATE statements in the program use addresses set for the 60-column text font. If you run it using the 80-column font, the labels won't show up in the right places.

```
SCREEN 1, 640, 200, 3, 2
WINDOW 2, "Shapes ", , 15, 1
```

```
MakeCircle:
  CIRCLE(160,50), 100, 5
  PAINT(160,50), 5, 5
```

(continued)

```

MakeTriangle:
  COLOR 6
  AREA(480,5)
  AREA STEP(120,90)
  AREA STEP(-240,0)
  AREA FILL

MakeRectangle:
  LINE(20,110) - (600,180), 4, BF

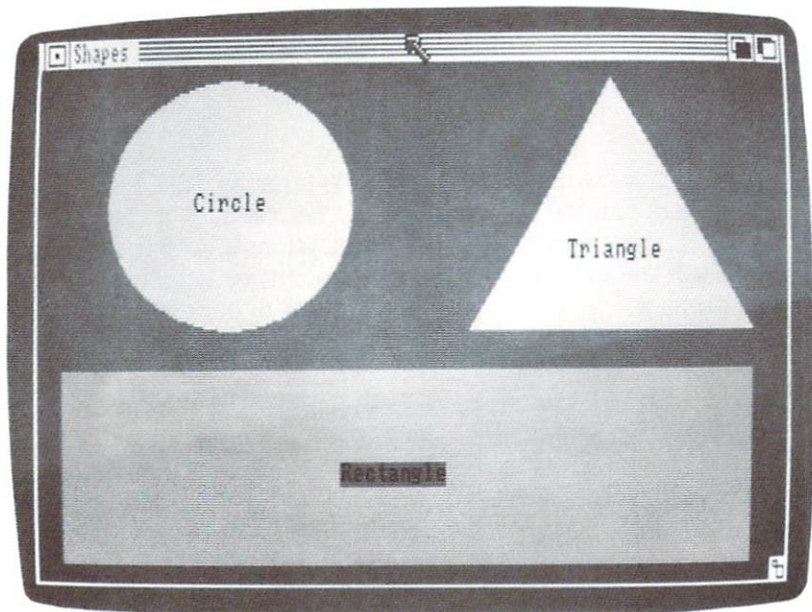
AddLabels:
  LOCATE 6, 14
  COLOR 2, 5
  PRINT "Circle"
  LOCATE 8, 45
  COLOR 2, 6
  PRINT "Triangle"
  LOCATE 17, 26
  COLOR 2, 0
  PRINT "Rectangle"

```

You can see the results in Figure 6-4.

Figure 6-4.

Text using different background and foreground colors is added to shapes in the window.



AN INTRODUCTION TO INTERACTIVE GRAPHICS

The graphics you have created in examples up to this point all depend on a stable environment: The window stays the same size, the number of bit planes remains the same, and the screen resolution is set at one resolution. You can't always depend on stable conditions for graphics, though. Intuition, the Amiga's user interface, allows the user to move windows around the screen, change window size, and change the graphics environment in many ways. When the environment changes, your program should be able to create interactive graphics—graphics that change in response to their environment.

Two Amiga BASIC functions—WINDOW() and POINT—give you the ability to read graphics conditions. You can use the results of these functions to create a program with graphics that change to fit the changing conditions; for example, the size of the figures might change to fit the size of the window, or the program might determine which of several windows has been selected with the pointer, and start creating graphics in that window. You can also use the results to create a subroutine that will work with many different resolutions and window sizes. Then you can store that subroutine and use it in different programs to perform the same task, confident that it will work with different resolutions and window sizes.

You can also use these functions to simplify pixel-address calculations; by reading the current size of the window, you know instantly where the borders are, so you won't specify a pixel address outside the border. Reading the current window size can also save you time trying to figure out just how many pixels a sizing gadget and title bar add or subtract from the window's interior area.

THE WINDOW() FUNCTION

Don't confuse the WINDOW() function with the WINDOW statement. The WINDOW statement creates a window on a screen. The WINDOW() function reads nine different window conditions and returns them to the program. The function also has an entirely different format:

```
WINDOW(condition number)
```

The condition number can be any integer from 0 to 8, and determines which condition WINDOW() will read. The WINDOW() function does no work in a program line by itself; it returns a value that should be assigned to a variable or used in a statement.

The following list gives the nine condition numbers followed by the condition that they read:

Condition number	Value returned
0	The window ID number of the window that's been selected with the pointer. The selected window is the window whose title bar is highlighted; all the other windows have ghosted title bars.
1	The window ID number of the current output window where PRINT and graphics statements do their work.
2	The width in pixels of the interior of the current output window.
3	The height in pixels of the interior of the current output window.
4	The column number of the pixel address in the output window of the bottom left corner of the text cursor. You can use this value along with the value for condition number 5 to exactly locate text in the window.
5	The row number of the pixel address in the output window of the bottom left corner of the text cursor.
6	The highest color-register number available in the output window. You can use this value to tell how many colors are available in the window.
7	The pointer to the current Intuition window. This pointer is a memory address used by advanced programmers in assembly-language subroutines.
8	The pointer to the current Intuition Rastport. This pointer is a memory address used by advanced programmers in assembly-language subroutines.

WINDOW() FUNCTION EXAMPLES

Since WINDOW() is a function and doesn't do much by itself, the best way to get a feel for its use is to try it in programs. The following program creates three different windows, numbered 2, 3, and 4, on the screen and sets window number 4 as the output window, since it was the last window created. It then prints out the number of the current output window and the currently selected

window, using WINDOW(1) and WINDOW(0) to read those values. If you move the pointer around the screen while the program is running and select different windows, you'll see the number of the selected window change.

```
WINDOW 2, "Window 2 ", (0,0) - (280,80), 15
WINDOW 3, "Window 3 ", (320,0) - (600,80), 15
WINDOW 4, "Window 4 ", (0,100) - (280,180), 15
PRINT "Output window is";
x1 = POS(0): y1 = CSRLIN
PRINT
PRINT "Selected window is";
x2 = POS(0): y2 = CSRLIN

Loop:
  LOCATE y1, x1
  PRINT WINDOW(1)
  LOCATE y2, x2
  PRINT WINDOW(0)
  GOTO Loop
```

You can see how this program looks in Figure 6-5.

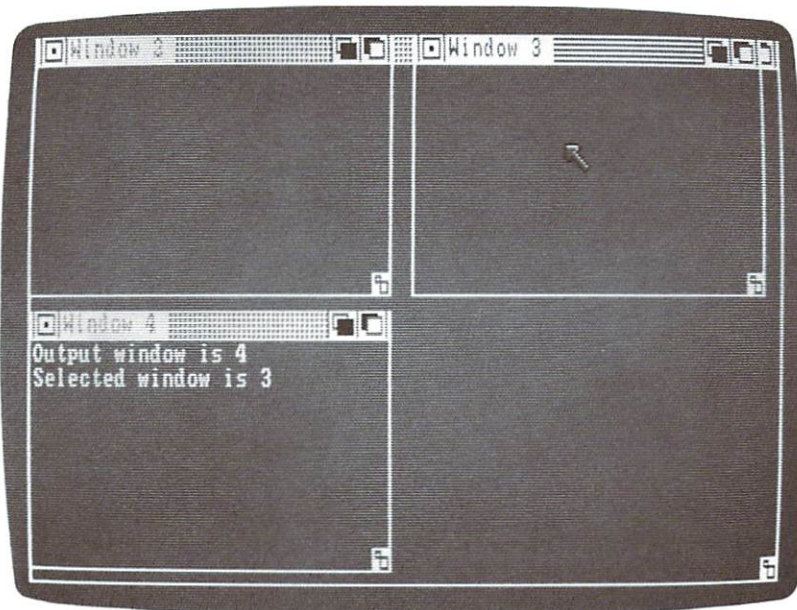


Figure 6-5.

The WINDOW(0) and WINDOW(1) functions show the output and selected windows.

The next program uses WINDOW(2) and WINDOW(3) to return the size of the output window and WINDOW(6) to return the number of colors the window supports. The first 12 lines of the program create a screen and an output window, then create an area pattern to

make the graphics that will follow interesting. The 13th line ties the timer to the RND function to make it truly random. The last 11 lines are a continuous loop that fills the output window with 18-sided AREA FILL figures.

In the first two lines of the loop, two WINDOW(6) functions return the number of colors available, which are modified by the RND and INT functions to return two random integers from 0 to the highest color-register number available. The COLOR statement uses these two integers to set random foreground and background colors.

The FOR . . . NEXT loop that follows sets 18 random points using the AREA statement. The address of the random points is set using the WINDOW(2) and WINDOW(3) functions so that the address will never fall outside the window boundaries. The AREA FILL statement that follows the FOR . . . NEXT loop connects the points and fills in the figure, and the GOTO statement after that starts the loop over again to create another random 18-sided figure. In Figure 6-6, you see an example of the random graphics this program generates.

When you run this program, try changing the size of the window with the sizing gadget. When the window changes size, the random figures created in it will adapt their size to reflect the new size of the window.

```
SCREEN 1, 640, 200, 4, 2
WINDOW 2, "Sizing Graphics ", , 15, 1
DIM area.pat%(7)
area.pat%(0) = $H3F3F
area.pat%(1) = $H9F9F
area.pat%(2) = $HCFCF
area.pat%(3) = $HE7E7
area.pat%(4) = $HF3F3
area.pat%(5) = $HF9F9
area.pat%(6) = $HFCFC
area.pat%(7) = $H7E7E
PATTERN , area.pat%
RANDOMIZE TIMER
```

Loop:

```
    forecolor = INT((WINDOW(6) + 1) * RND(1))
    backcolor = INT((WINDOW(6) + 1) * RND(1))
    COLOR forecolor, backcolor
    FOR i = 1 TO 18
        x = INT(WINDOW(2) - 1) * RND(1)
        y = INT(WINDOW(3) - 1) * RND(1)
        AREA(x,y)
    NEXT i
    AREA FILL
    GOTO Loop
```

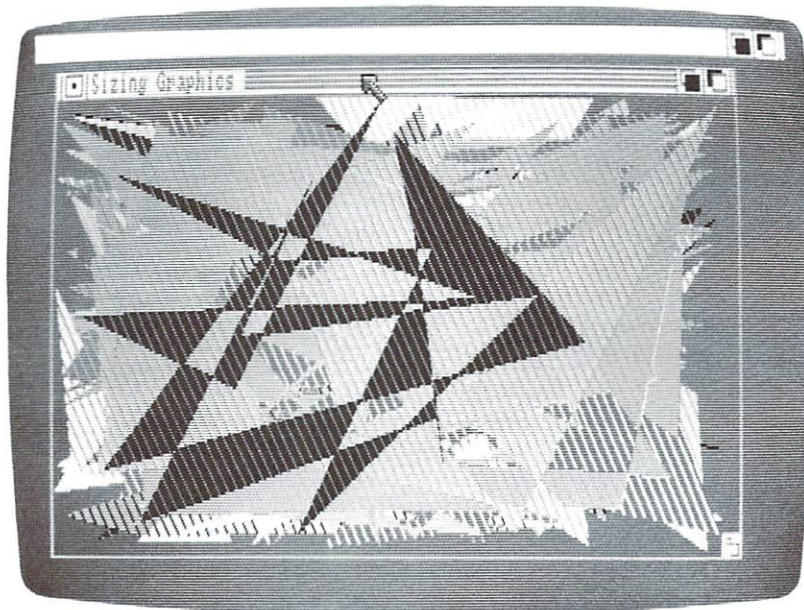


Figure 6-6.

The WINDOW() function sets the size and colors of these shapes to fit within the window they're drawn in.

This last WINDOW() function example is a subroutine that takes any text string, along with the text row and column position where you want it to be printed, and prints the text in a box. WINDOW(4) and WINDOW(5) return the pixel addresses of the beginning and ending locations of the text cursor. The LINE statement then draws a box around the text using these addresses, plus or minus a few pixels up, down, right, and left to adjust for extenders (like the tail on the p) and to provide clearance over the top of the characters. The first four lines of the program create the string, set the address of the text cursor, and then call the subroutine, *BoxIt*. You can see the results of the program in Figure 6-7 (on the next page).

```

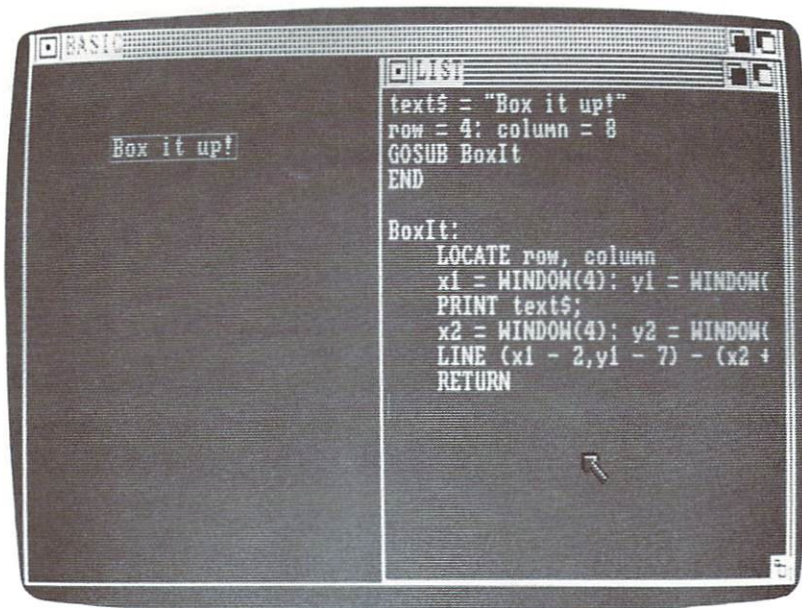
text$ = "Box it up!"
row = 4: column = 8
GOSUB BoxIt
END

BoxIt:
  LOCATE row, column
  x1 = WINDOW(4): y1 = WINDOW(5)
  PRINT text$;
  x2 = WINDOW(4): y2 = WINDOW(5)
  LINE(x1 - 2,y1 - 7) - (x2 + 1,y2 + 2), 3, B
  RETURN

```

Figure 6-7.

WINDOW(4) and WINDOW(5) locate the start and end of a text string to allow you to draw a box around it.



THE POINT FUNCTION

The POINT function looks at any pixel in the output window and returns the number of the color register used to color that pixel. Its format is simple:

```
POINT(pixel address)
```

The pixel address can't be a relative address, it can only be an absolute address.

The following short program creates three different colored areas on the output window. In a FOR ... NEXT loop, it then locates the text cursor inside each of the areas and uses the POINT function to read the color and set the foreground and background color of the text so it will show up in the area when it's printed:

```
LINE(0,0) - (620,60), , BF
LINE(0,61) - (620,120), 2, BF
LINE(0,121) - (620,180), 3, BF
FOR i = 0 TO 2
  LOCATE(7 * i) + 3, 5
  background = POINT(WINDOW(4), WINDOW(5))
  COLOR background - 1, background
  PRINT "Popocatepetl"
NEXT i
```


COPYING AND PASTING GRAPHICS

There are times when you will want to copy graphics from one section of the output window to another section of the output window without redrawing them. Amiga BASIC has two statements—GET and PUT—that allow you to copy a square section of any window and move it to another window or to another location in the same window.

THE GET STATEMENT

You use the GET statement to specify the square section of the output window, called a graphics block, that you want to copy. You also use GET to specify the name of the variable array you want to use to store the block in. The format for GET is:

```
GET opposing corner addresses, array name(index number)
```

Like the box options of the LINE statement, GET needs two opposing corner pixel addresses to define a graphics block. They can only be absolute addresses, not relative addresses. They're enclosed in parentheses and separated by a dash; for example, (30,0) – (60,40) defines a block 31 pixels wide and 41 pixels high. The array name is the name of a variable array that you use to store the contents of the graphics block. It's followed by an optional index number in parentheses; the index number is explained later in this chapter.

Creating a variable array for graphics-block storage

Before you use a GET statement, you have to create a variable array to store the graphics data from the graphics block. You create the array by setting its size with a DIM statement. You can use any kind of variable for the array except a string variable, but the best array to use is a short integer array, specified by adding a % sign at the end of the variable name.

The size of the graphics block determines the size of the array—the larger the graphics block, the larger the array has to be to store the block. A graphics block is measured in three dimensions: width, height, and depth. The width and height are measured in pixels. The depth is determined by the depth of the screen the graphics block is on, and is measured in bit planes. As an example, consider a graphics block stretching from pixel (25,3) to pixel (240,76) on a screen three bit planes deep. The dimensions of the graphics block are 216 pixels wide (column 25 to column 240, inclusive) by 74 pixels high (row 3 to row 76, inclusive) by 3 bit planes deep (the depth of the screen).

A short review of arrays might help you understand how GET uses an array to store the graphics block. When you first create an array with a DIM statement, you set the number of elements in the array. For example,

```
DIM graphics%(34)
```

creates an array named *graphics%* with 35 elements numbered 0 to 34. Each array element is a variable that can store data, and is specified by the index number in the parentheses. For example, *graphics%(0)* is the first element in the array, *graphics%(1)* is the second element, *graphics%(2)* the third element, and so on.

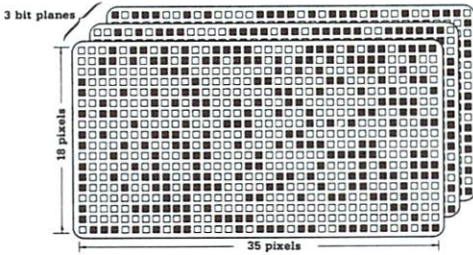
How the array stores graphics data

The short integer array you use to store a graphics block uses two bytes of RAM to store the contents of each element of the array. Each element can store a number as low as -32768 or as high as 32767 , or it can store 16 bits of graphics data from the GET statement. When GET uses the array to store a graphics block, it uses the first three elements (0, 1, and 2) to store the dimensions of the graphics block. The first element of the array stores the width of the block, the second element stores the height of the block, and the third element stores the depth of the block.

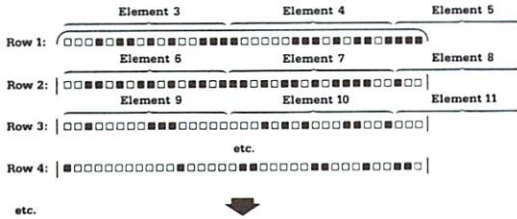
The rest of the array stores the graphics data of the graphics block. It stores them by bit planes, starting with the first row of bits in the first bit plane and working down to the last row of bits in the first bit plane, and then does the same thing for each of the bit planes that follow. Figure 6-8 shows you how the bit planes are stored in the array elements.

Notice that GET doesn't break up an element at the end of a bit-plane row. If there aren't enough bits to fill up the last element of the row, it just fills it partially and fills the rest of the element with zeros. For example, it takes three elements to store a bit row of a graphics block that's 35 pixels wide. The first two array elements store 32 of the bits in the row, and the third element stores 3 bits—its other 13 bits are tacked on the end as zeros.

Graphics block



Bit plane 1



Element #:	Array contents:
Width: 0	35
Height: 1	18
Depth: 2	3
Row 1, Bit plane 1	0001011010100111
4	1000001110101101
5	1110000000000000
6	0011010110101101
Row 2, Bit plane 1	1101011010111100
8	1000000000000000
Row 1, Bit plane 2	0010101110000111
58	1011011111001111
59	1100000000000000
Row 18, Bit plane 3	1101011100001011
164	0010110111011000
165	1110000000000000

Figure 6-8.

How an array stores the graphics data of a graphics block.

Calculating the size of the array

To figure the size of the array you need to store a graphics block, first figure out how many array elements you need to store one bit-plane row of the block. To get that figure, just divide the width of the block in pixels by 16, which gives you the number of pixels each element of a short integer array will hold. If you have any leftover pixels, add one element to the results. For example, if you have a block that's 35 pixels wide, the array needs 3 elements: 35 divided by 16 yields 2 with a remainder of 3. The remaining three pixels need an entire array element for storage, so the total number of elements needed for one line is 3.

Next, multiply the number of elements for one bit-plane row by the height and depth of the graphics block. This gives you the number of elements you need to store the graphics data for the entire block. Add three more elements to store the three dimensions of the block, and you have the total size of the array you need to store the block.

Try an example: If you're going to store a graphics block that stretches from (36,5) to (105,72) in a 16-color (4-bit-plane) screen, you have a block that's 70 pixels wide by 68 pixels high by 4 bit planes deep. It takes 5 array elements to store one line (70 divided by 16 yields 4 with a remainder of 6, so it takes 5 elements). Multiply that by the height (68) and the depth (4), and you get 1360. Add 3 to store the dimensions of the block, and you get a total of 1363 dimensions. You use the statement

```
DIM block%(1362)
```

to create an integer array named *block%* with exactly 1363 elements. (Each array always starts with element number 0, so if you dimension it up to element 1362, as you did in the statement before, you actually get 1363 elements in the array.)

Storing the graphics block in the array

Once you create the array, you can follow it with a GET statement to copy the graphics block into the array. Just follow the opposing corner addresses with a comma and then the name of the array:

```
GET(36,5) - (105,72), block%
```

Copying several equal-sized graphics blocks in one array

If you want to copy several graphics blocks of the same size, you can avoid the work of creating an array for each block by storing all the blocks in one array. When you dimension the array, first figure out the size you need for one block using the method described before. Then dimension your array as a two-dimensional array, using the size of one block as the first dimension and the number of blocks you want to store as the second dimension.

For example, to copy five different blocks, each the size of the block used in the last example, each block needs 1363 elements for storage. You would create an integer array that measures 1363 elements by 5 blocks with this statement:

```
DIM block%(1362,4)
```

Remember that since each index number starts at 0, *block%* actually measures 1363 elements by 5 blocks even though the DIM numbers are one smaller.

When you save the five graphics blocks with five GET statements, the two index numbers change for each block. The first block is stored starting at *block%(0,0)* and ending at *block%(1362,0)*. The second block starts at *block%(0,1)* and ends at *block%(1362,1)*. And so it goes until the fifth block is stored starting at *block%(0,4)* and ending at *block%(1362,4)*.

To point to the beginning of the data for each graphics block, you just set the first index number of the array to 0, then set the second index number to the number of the block (0 to 4). This makes it very easy to use different GET statements to store different blocks with the same variable. For example, these five GET statements store five different graphics blocks in five different parts of the *block%* array:

```
GET(36,5) - (105,72), block%(0,0)
GET(46,75) - (115,142), block%(0,1)
GET(136,5) - (205,72), block%(0,2)
GET(236,5) - (305,72), block%(0,3)
GET(36,105) - (105,172), block%(0,4)
```

Copying several unequal-sized graphics blocks in the same array

If you want to store several graphics blocks of unequal size in the same array, you use a one-dimensional array just as you would to save a single graphics block. To dimension the array, you must first figure out the number of elements each block needs for storage, using the same method you use for a single block. Then add all the values together to come up with the total number of elements you need to store all the blocks together. When you store the blocks, you store them in the array one after the other.

As an example, consider using GET to store three different blocks, all on a 3-bit-plane screen. The first block stretches from (0,0) to (158,40), the second from (26,75) to (310,146), and the third from (50,150) to (74,174). When you figure the number of array elements needed to store each block, you come up with 1233 elements for the first block, 3891 elements for the second block, and 153 elements for the third block. Adding all the elements together, you find you need an array with 5277 elements, so you use

```
DIM block%(5276)
```

to create one.

When you copy the three graphics blocks to the array, you use elements 0 to 1232 for the first block, which requires 1233 elements. You use elements 1233 to 5123 for the second block, which requires

3891 elements, and you use elements 5124 to 5276 for the third block, which requires 153 elements. Note the starting-element index number for each block so you can use it with three different GET statements when you copy the three blocks. For example, these statements save the three example blocks, starting at index number 0 for the first block, index number 1233 for the second block, and index number 5124 for the third block:

```
GET(0,0) - (158,40), block%(0)
GET(26,75) - (310,146), block%(1233)
GET(50,150) - (74,174), block%(5124)
```

You can store as many different blocks in one array as you wish, as long as you dimension the array large enough and you don't run out of memory.

THE PUT STATEMENT

The PUT statement takes the image stored in an array by GET and puts it in the output window at the location you specify. It uses this format:

```
PUT address, array name(index number), merge choice
```

The address can be an absolute or a relative pixel address. The array name is the same one you used to store the graphics blocks with the GET statement. The index number is optional, you use it only if you saved more than one graphics block in the array. The merge choice is also optional; it's one of five words that specify how the graphics block merges with graphics that it may cover in the window.

Setting the PUT address

The address you include in the PUT statement specifies where the upper left corner of the graphics block will appear in the output window. You can use any address in the window. If you specify an address that runs some of the graphics block out of the window, PUT trims off any part that goes beyond the window's boundaries.

The array name and index

When you specify an array name for PUT, use the same array name you used to save the graphics block using GET. If you didn't use an index number with GET, you won't need one in PUT. For example, if you saved a graphics block in the array *swath%*, then the statement

```
PUT(0,0), swath%
```


will put the graphics block stored in *swath%* in the window with the upper left corner of the block located at pixel (0,0).

If you saved several blocks in one array and used different index numbers to mark the beginning of each block, then use the same array name and index number for each block that you used in the GET statement. For example, to display the five equal-sized blocks you saved earlier in the array named *block%*, use these five PUT statements:

```
PUT(0,0), block%(0,0)
PUT(80,0), block%(0,1)
PUT(160,0), block%(0,2)
PUT(240,0), block%(0,3)
PUT(0,100), block%(0,4)
```

You can copy the same block to different locations in the window as many times as you want. Just use the same array name and index with different addresses. For instance, these four statements copy the graphics block stored in *swath%* to four different locations in the output window:

```
PUT(0,0), swath%
PUT(50,50), swath%
PUT(110,100), swath%
PUT(204,0), swath%
```

The merge choice

There are five merge choices you can add to the PUT statement to affect the way the graphics block merges with the graphics underneath the block in the output window. They are:

- **PSET**
Puts the graphics block in the window, covering up any graphics that lie beneath it.
- **PRESET**
Turns every color in the graphics block to its inverse color, and then puts the graphics block in the window, covering up any graphics that lie beneath it.
- **AND**
Performs the Boolean AND operation between each pixel in the graphics block and the pixel in the output window that it covers up and displays the result. It's easiest to understand if you concentrate on the results, rather than the process. Wherever there is background color (color-register 0) in the output window, AND won't lay down any of the contents of the

graphics block. Wherever there is the last possible color of the window (the highest possible color-register number), AND puts down the colors of the graphics-block just as they are. Any other colors in the output window combine with the graphics block colors to create new colors predictable only by using Boolean AND algebra.

- **OR**

Performs the Boolean operation OR between each pixel of the graphics block and the pixel in the output window that it covers. OR lays down the graphics block in its original colors wherever there is background color, and doesn't lay down anything where the last possible color of the window (the highest color-register number) is. Any other colors in the graphics block combine with colors in the output window to create new colors predictable only by using Boolean OR algebra.

- **XOR**

Performs the Boolean operation XOR between each pixel of the graphics block and the pixel in the output window that it covers. XOR lays down the graphics block in its original colors wherever a background-color pixel in the output window is covered with a graphics-block pixel. Wherever a background color in the graphics block covers an output window pixel, XOR lays down the output window pixel color. Wherever two non-background colors are merged, a new color that is neither of the two merging colors results.

If you don't specify a merge choice, PUT automatically assumes that the merge choice is XOR.

XOR has a very useful property. If you PUT the same graphics block in the same location twice using XOR, the first time you do it the block appears in the colors determined by the XOR merge choice. When you use XOR the second time, the block put there by the first XOR statement will disappear completely, and the graphics underneath it will appear exactly as they were before the PUT statements, effectively erasing the first block.

Applications for PSET and PRESET are obvious: You can use them to insert graphics blocks and inverse graphics blocks wherever you want them. Applications for AND and OR are not quite as obvious, but you can use them with a background mask, which you use like a painter uses a stencil. The cutouts in a paper

stencil define shapes, such as block letters, that you can paint in—but you can't paint in the surrounding area because the stencil paper covers it up; a background mask similarly defines a shape in a window that the PUT statement using AND and OR must use when it lays down a graphics block.

You can create a mask by creating a shape in the output window using the highest possible color register and then surrounding it by background color. When you lay a graphics block over the mask using the AND merge option, the mask will be filled with the colors in the graphics block, and the surrounding area won't be affected by the graphics block. A mask works much the same way with OR, but instead of filling the mask with the graphics block, OR leaves it alone and fills the surrounding area with the block instead. You can see an example of masking in the program that follows later in this chapter.

You can see the results of all five merge choices in Figure 6-9 later in this chapter.

A PROGRAM EXAMPLE USING GET AND PUT

In the following program, you can see five different graphics blocks saved in a two-dimensional array, then laid back down in different locations in the same window using the five merge options. The first eleven lines of the program create a window in a screen that's four bit planes deep, then use two FOR . . . NEXT loops to fill in the upper part of the window with words and background in all the colors possible in the window. The next eight lines print text and draw circles in the bottom half of the screen using the highest color register (number 15, a light gray) against the background color.

The DIM statement in the next line creates a two-dimensional short integer array that measures 803 by 5 elements. This is to store five blocks that each need 803 elements of array storage. The FOR . . . NEXT loop that follows cycles five times, and each time uses the GET statement to store a block in the top section of the screen. The following LINE statement outlines the area that the GET statement saved so you can see where it copied graphics. A FOR . . . NEXT loop in the next line freezes the display for a short time so you can see how it looks before the graphics blocks are PUT back in the window.

The last five lines of the program are five different PUT statements that lay the five blocks down in a row on the bottom of the screen. Each PUT statement uses a different merge choice.


```

SCREEN 1, 320, 200, 4, 1
WINDOW 2, , , 0, 1
WIDTH 40
FOR i = 1 TO 3
  FOR j = 0 TO 15
    k = j + 1
    IF k > 15 THEN k = 0
    COLOR j, k
    PRINT "Trinidad";
  NEXT j
NEXT i
LOCATE 16, 1
PRINT "and the big Mississippi and the";
PRINT "town Honolulu and the lake";
PRINT "Titicaca."
FOR i = 35 TO 287 STEP 63
  CIRCLE(i,170), 10
  PAINT(i,170)
NEXT i
DIM block%(802,4)
FOR i = 0 TO 4
  GET(60 * i,0) - ((60 * i) + 59,49), block%(0,i)
  LINE(60 * i,0) - ((60 * i) + 59,49), 1, B
NEXT i
FOR t = 1 TO 5000: NEXT t
PUT(5,130), block%(0,0), PSET
PUT(68,130), block%(0,0), PRESET
PUT(131,130), block%(0,0), AND
PUT(194,130), block%(0,0), OR
PUT(257,130), block%(0,0), XOR

```

You can see the results in Figure 6-9.

Notice how the different merge choices affect the five blocks laid down across the bottom of the screen. The first block, using PSET, is copied directly to the window, covering everything underneath it. The second block, using PRESET, is converted to inverse colors and laid down in the window covering everything beneath it. In the third block, the letters and the circle act as a mask, since they are colored with register number 15, the highest color register; the colors in the block are copied within the characters and the circle, but not outside them in the background color. In the fourth block, the opposite happens. The block colors are copied in the background color all around the characters and the circle, but the characters and circle themselves are unchanged and show through the block. In the last block, the characters are changed to completely new colors by XOR, depending on which color in the block covers the character.

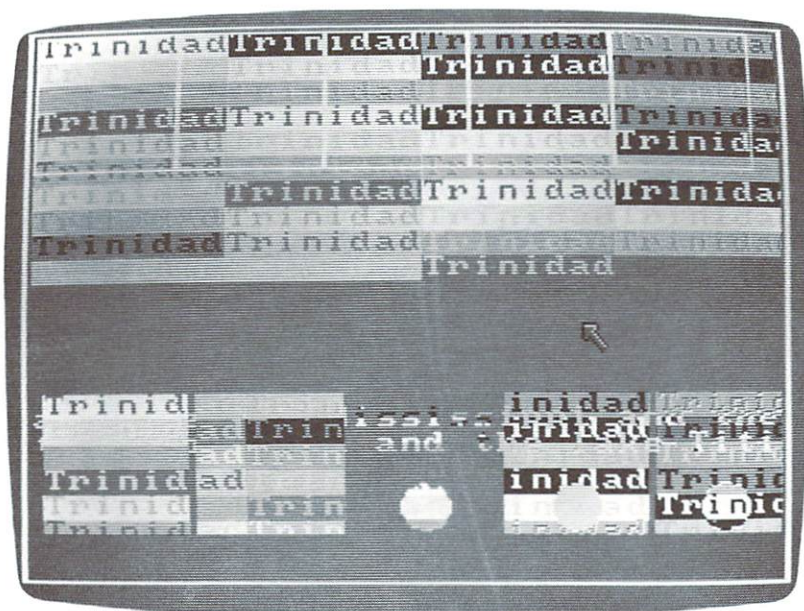


Figure 6-9.

GET and PUT statements copy blocks from one section of the window to another section using the five merge choices.

You've now tried some of the more exotic Amiga BASIC graphics commands, and have added text to your graphics using different fonts and character colors. You've also created interactive window graphics that respond to changing window conditions. With the GET and PUT statements, you've copied and pasted blocks of graphics from one section of a window to another, and learned how to store multiple graphics blocks in one array. This knowledge will come in handy when you use PUT to create animation sequences—but that's not until Section 4, Animation. In the meantime, turn up your monitor speaker and try some Amiga sounds in the section ahead.

SECTION 3

Sounds

In this section, you'll learn how to create music and speech with the Amiga. In Chapter 7, you'll learn the fundamentals of electronic music. In Chapter 8, you'll learn advanced techniques for using the Deluxe Music program, and learn about other music applications and additional hardware that will enhance the Amiga's sound capabilities. Chapter 9 will show you how to use Amiga BASIC to create music, sound effects, and speech.



**CHAPTER SEVEN
AN ELECTRONIC
MUSIC PRIMER**

Although most people hear electronic music all the time on radio, television, and at the movies, if you ask them what created the music, most of them aren't quite sure. They see visions of giant consoles of knobs, switches, and glowing lights, and rat's nests of patch cords that look like the cover of *Switched On Bach*, or they see a rock band with huge racks of keyboards and cables running everywhere. These are visions of electronic music's sordid past.

Computers have changed electronic music for the better, making it simpler and much more fun to create. There's no longer any need for the massive banks of switches and knobs—the computer can offer you the same control using simple commands issued from the keyboard or mouse. The cabling is much simpler because there aren't so many components needed to create music. And if you use the Amiga to create music, the process is very simple indeed, because all the necessary hardware is included in the machine—all you need is the software to control it.

Since the Amiga can perform so many different audio functions, it's a good idea to get to know about them individually—how they work, and what they're supposed to do. The aim of all this audio-technical wizardry is to please your ears, so it's also good to know just how your ears are tickled into hearing sounds. The more you know, the better sounds you can make with your Amiga.

HOW YOU HEAR SOUNDS

Most people know that sound comes to your ears through the air. After all, if you cover your ears with your hands, you can cut out a lot of sound. It's also obvious that sound usually comes from a variety of sources around you. In a room full of people talking, you can usually tell which person is speaking (at least if you're close enough). What most people don't know is how it all works—why some sounds are different than others, and how your ears interpret each sound.

Sounds begin with a vibrating object like a gong, vocal cords, or a plucked string. As the object vibrates forward, it pushes and slightly compresses the air around it. As the object vibrates back, it pulls back and rarefies (creates a slight vacuum in) the air. As it continues vibrating, it creates more compressions and rarefactions of the air. These pressure variations travel away from the object as sound waves, as you can see in Figure 7-1. They spread through the air much like ripples spread out from a rock thrown into a still pond, and travel until they die out or hit another object (such as your ear).

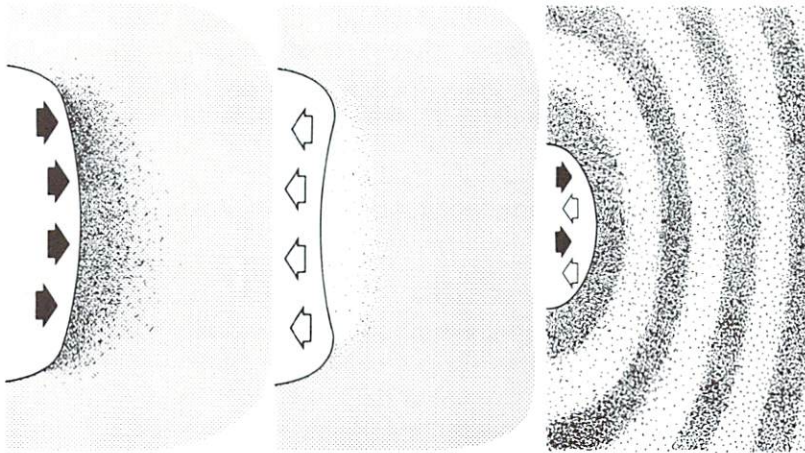


Figure 7-1.

In the drawing at left, a vibrating object vibrates out and compresses the air beside it. In the middle drawing, the same object vibrates back in, rarefying the air beside it. The drawing at right shows these compressions and rarefactions spreading out from the object as sound waves.

INSIDE THE EAR

There are three parts to a human ear: the outer ear, which you can see sticking out from the side of your head, the middle ear, and the inner ear, tucked deep inside your skull. Figure 7-2 shows these three parts. The flesh and folds of the outer ear gather and concentrate sound waves and pass them on to the middle ear. Here, the sound waves meet the eardrum, a small membrane that transmits vibrations to three small bones (called the hammer, anvil, and stirrup), which amplify the sound waves and pass them on to the cochlea, a part of the inner ear.

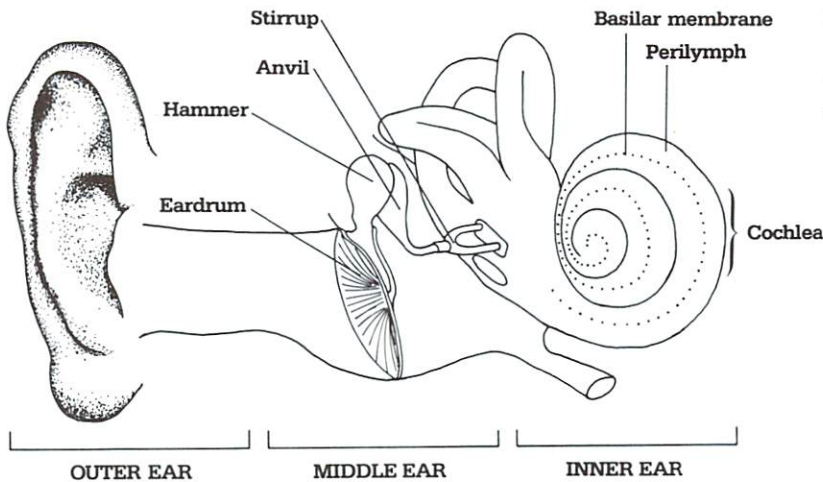


Figure 7-2.

The three parts of the ear.

The cochlea is a cavity in the bony part of the skull that's coiled like a snail shell. It's filled with a liquid called perilymph, and divided along its coiled length with a very sensitive membrane called the basilar membrane. The amplified vibrations of the eardrum pass on to the perilymph, which makes the basilar membrane vibrate. Nerve endings on the basilar membrane send the sensations of vibration on to the brain, which analyzes them so you can hear sound.

CHARACTERISTICS OF SOUND

When your brain analyzes sound, it can discern four main characteristics of vibration. They are frequency, amplitude, timbre, and duration. Using these four characteristics, you can describe any sound you hear, or use them to provide the specifications for creating a new sound.

Frequency

Frequency is the speed of vibration. When an object vibrates slowly, it creates a low-frequency sound; when it vibrates quickly, it creates a high-frequency sound. The frequency of a sound determines its pitch—the higher the sound's frequency, the higher the pitch.

As an example of different frequencies, think of plucking the strings on a guitar. Because the thicker strings are heavier than the thin strings, they vibrate more slowly and therefore sound lower in pitch than the thin strings.

Frequency is measured in hertz, abbreviated Hz, a unit that stands for cycles (vibrations) per second. The average range of human hearing stretches from 20 Hz to 20,000 Hz.

Amplitude

Amplitude is the strength of vibration. When an object vibrates with strong vibrations (that is, it vibrates back and forth over a relatively large distance), it creates a sound of strong amplitude. When it vibrates with weaker vibrations (vibrating back and forth over a relatively small distance), it creates a sound of weaker amplitude. The amplitude of a sound determines its volume—the higher the amplitude, the louder the sound. As an example of amplitude, think of the guitar strings in the previous example. If you pluck them hard, they vibrate violently, creating a sound of great amplitude. If you pluck them gently, they vibrate so little you can barely see them move, making sounds of little amplitude.

The amplitude of a sound is measured in decibels. A decibel is the smallest change in loudness that a human ear can detect. Increasing a sound's amplitude by ten decibels makes the sound double in loudness to the human ear. Human hearing ranges from one decibel (the threshold of hearing) to somewhere over 120 decibels (the threshold of pain).

Timbre

Timbre (pronounced tam'-burr) is a little more complicated than frequency and amplitude. It is a mixture of frequencies within a single sound. Most vibrating objects don't vibrate at just one frequency, they vibrate at several frequencies simultaneously. The lowest frequency is called the fundamental, and it's the fundamental that you hear as the main frequency of the sound. The fundamental sets the pitch of the sound. The higher frequencies are called overtones, and they blend in with the fundamental frequency to change the tonal quality of the sound.

The timbre of a sound determines its tone color. The more overtones present in the sound, the richer its timbre; the fewer the overtones, the thinner its timbre. Timbre isn't measured in any unit of measurement, although it is possible to analyze the number of overtones present in a sound (a process called Fourier analysis) to see how many overtones are present. Instead, most people use words like "rich," "thin," "fat," or "buzzy" to describe timbre.

A good example of timbre is the difference between a violin and a flute playing exactly the same note. Each instrument produces sounds with an entirely different overtone series, so each has its own distinct timbre, and won't be mistaken for the other.

Although Fourier analysis can show the individual overtones of a sound, the most common way to show the timbre of a sound visually is to use a waveform. A waveform is a record of the air pressure of a sound wave over time that shows the summation of sound's overtones. By looking at the shape of the cycles in the waveform, you can see differences in timbre. In Figure 7-3, you can see three basic wave shapes: the sine wave, the square wave, and the sawtooth wave. The sine wave sounds gentle and smooth. The square wave is rich and full, and the sawtooth wave is quite piercing and colorful.

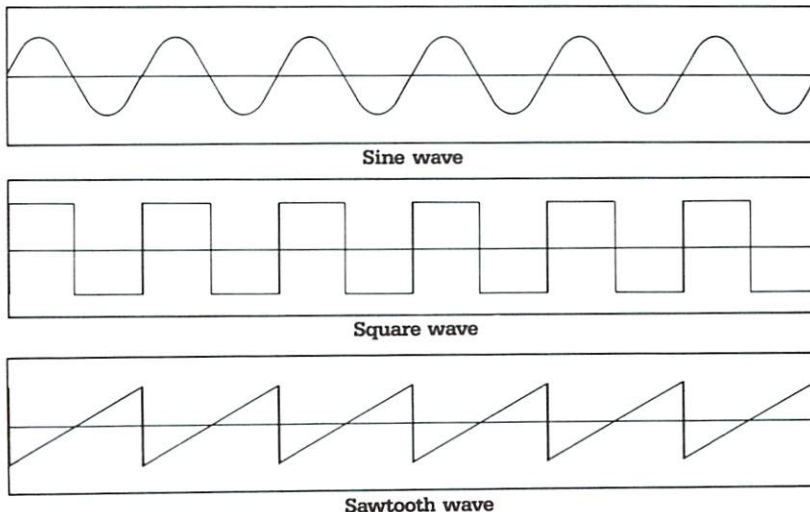


Figure 7-3.

Three waveforms: a sine wave, a square wave, and a sawtooth wave.

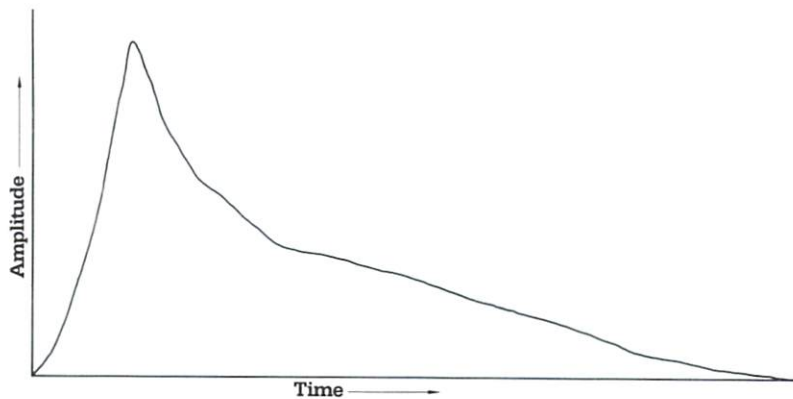
Duration

Duration is used to measure the first three sound attributes: frequency, amplitude, and timbre. In its simplest use, duration measures all three together, and so measures the length of the entire sound from its first hearing to its last fadeout. In other cases, duration measures each attribute individually, recording how it changes over time. For example, a singer can hold out a note on one pitch, but get louder and then fade away. Duration can measure how long the amplitude increased and how long the amplitude decreased. A trumpet player might play a long note on the same pitch, but stick a mute in halfway through the note. Duration could be used in this case to measure the length of the first timbre before it was replaced by the muted timbre. If a saxophone player squeaked at the beginning of a note, duration could measure the time it took the squeak frequency to drop back down to the frequency of the final note.

The graph of any of these three attributes over time defines the envelope of that attribute. Envelopes are very important in describing the quality of a sound. For example, a plucked harp string has a sharp twang at the beginning that dies away as the note lingers. Looking at the amplitude envelope of that note in Figure 7-4 shows that the note immediately jumps to large amplitude (loud), then slowly decreases its amplitude over time (gets softer and fades away).

Figure 7-4.

The amplitude envelope of a harp note.



Envelopes are very important to the way we hear sounds. Any sound is a result of the way its pitch, loudness, and timbre are shaped over time. Changing the envelope of any one attribute makes a distinct change in the character of the sound.

PRODUCING ELECTRONIC SOUNDS

To hear sounds from an electronic source, such as a tape deck, synthesizer, or a computer, you need a sound system consisting of at least two components: a speaker, and an audio amplifier to drive the speaker. The amplifier takes a weak audio signal from an electronic device, amplifies it, and sends the signal to the speaker, where the speaker does the physical work of converting the amplified audio signals into audible form. The speaker has the difficult task of creating a huge variety of sounds—sounds with many different timbres, sounds over a wide range of pitch and loudness.

THE SPEAKER

If you look at Figure 7-5, you can see a cutaway drawing of a speaker. The vibrating part of the speaker, the part that produces sound waves, is the speaker cone. A speaker cone is a conical piece of stiff paper or plastic mounted in a circular metal frame. The outer edge of the cone is suspended in the frame by very flexible folds of rubber or rubber-like material so the cone can move in and out easily within the metal frame.

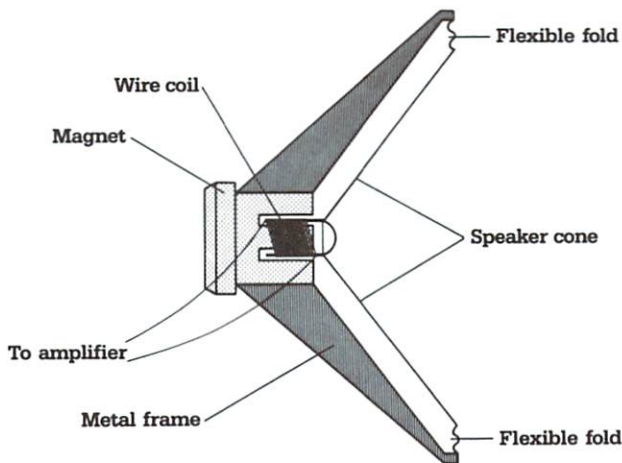


Figure 7-5.

A cutaway view of a speaker.

In a typical speaker, the center of the speaker cone is attached to a coil of wire suspended in the magnetic field of a very powerful magnet, which is attached to the rear of the metal frame. (This magnet is powerful enough to erase floppy disks, so you should always be careful not to put your disks too near the back of a speaker.) Both ends of the coil of wire lead out of the speaker to the amplifier.

When the amplifier sends an audio signal through the coil, it creates a magnetic field in the coil that either attracts or repels the magnetic field created by the fixed magnet. This pushes the coil forward or backward, moving the speaker cone with it. The direction and amount of the speaker-cone movement depends on the strength of the magnetic field created by the audio signal going through the coil. The stronger the field, the more the cone moves.

The audio signal coming from the amplifier changes the strength of the field very rapidly, so the speaker cone vibrates quickly enough to create sounds. As the audio signal changes in frequency, amplitude, and overtones, the sound produced by the speaker changes with it. A good speaker cone will accurately interpret every nuance of change in the audio signal as a nuance in the sound it creates.

In a good high-fidelity speaker, there are usually at least three speaker cones, one for each of three different frequency ranges. One speaker cone, appropriately called the woofer, produces low sounds. The mid-range speaker cone produces mid-range frequencies, and the tweeter is a speaker cone that produces high pitches. A system of filters, called the crossover network, separates the audio signal from the amplifier into three separate signals of the three different frequency ranges, which are sent out over separate wires to each speaker cone. The result, if the speaker components are of good quality, is excellent reproduction of all the frequencies in the human hearing range.

THE AMPLIFIER

An audio signal from sources like cassette decks, turntables, radio tuners, or the Amiga is too weak to drive a speaker by itself; it needs amplification in order to be heard. An amplifier's job is to take an audio signal from a weak source and boost its strength so that it's strong enough to move the speaker cone in and out. Almost all amplifiers have a volume knob so you can control how much the signal is amplified; some amplifiers also have tone controls that change the timbre of the incoming signal before it's amplified and sent out to the speaker.

STEREO

Most people have a stereo home-audio system with two speakers, one for the right of the listening area, the other for the left of the listening area. These speakers convert two separate audio signals (called the left and right channels) into two sound sources that create the illusion of many different sources of sound located throughout a room (a process called imaging).

To see how stereo imaging works, consider an example. A recording engineer records a woodwind trio spread across a stage: the oboe on the left, the clarinet in the middle, and the bassoon on the right. He uses two microphones to record, one for each channel. The left microphone is close to the oboe and further from the other instruments, and so records the left channel with a strong oboe sound, a medium clarinet sound, and a weak bassoon sound. The right microphone, close to the bassoon, records the right channel with a strong bassoon sound, a medium clarinet sound, and a weak oboe sound.

When you listen to the recording (facing the speakers with the left-channel speaker on your left and the right-channel speaker on your right), you hear a lot of oboe sound coming from the left speaker and not much oboe sound coming from the right speaker, so you perceive the oboe to be to your left. The clarinet sound comes equally from both speakers, so you perceive the clarinet coming from the center area between the two speakers. The bassoon sound comes mostly from the right speaker, so you perceive the bassoon to be on your right. Your stereo system uses this same imaging technique to create the illusion of an entire orchestra of instruments spread across your living room.

The Amiga has stereo outputs so you can connect it to a stereo system, but its left and right channels are discrete—that is, the left signal goes entirely to the left speaker, and the right signal goes entirely to the right speaker. These discrete signals don't allow imaging, since there is no mixture of the sound coming from both speakers. To blend the Amiga's sound signals through both left and right channels to create imaging, you need an audio mixer, described later in the chapter.

SYNTHESIZING MUSIC

To record music, you use microphones that turn sound waves in the air into audio signals. When you listen to the recording, the sound system turns audio signals back into sound waves. If you create music by building an audio signal from scratch rather than recording sound waves, you are *synthesizing* music.

Music synthesis requires more than just an amplifier and a speaker. You need a synthesizer to build an audio signal, and a controller like a keyboard or a sequencer, to play music on the synthesizer. The Amiga has the hardware necessary to synthesize sounds through its audio ports. You can attach a music keyboard to a controller port or use the Amiga's own keyboard to play the Amiga like a musical instrument. You can buy or write software that creates a sequencer in RAM to play sequences of notes using the Amiga's synthesizing capabilities.

SYNTHESIZERS

In concept, a synthesizer is simple. It sets the frequency, amplitude, and waveshape of an audio signal, and sends it as an unamplified audio waveform to the amplifier. The amplified waveform is then sent to the speaker, where it is translated directly into sound waves, audibly duplicating the original synthesized waveform. In practice, sound synthesis isn't quite so simple. It takes a lot of work to create an audio signal that's interesting to the human ear. The first synthesizers to succeed were analog synthesizers, instruments that are still popular today.

Analog synthesizers

Analog synthesizers use a series of electronic components first to create a simple audio signal, then to twist, tickle, and torture the signal into a much more complex and interesting form. The heart of the analog synthesizer is the voltage-controlled oscillator, or VCO for short. The VCO creates an audio signal from scratch in one of several waveforms that you can choose. These waveforms are usually quite simple—sine waves, triangle waves, sawtooth waves, square waves, and perhaps a few others.

The VCO changes pitch according to a control voltage, which usually comes from a keyboard or a sequencer. For each different note, the keyboard or sequencer sends a different control voltage to change the pitch of the VCO. To make the sound more interesting, the signal from the VCO can be passed through voltage-controlled filters and voltage-controlled amplifiers that add attributes such as overtones, duration, and amplitude to each note. The signal can be enhanced even more by passing it through other synthesizer modules: ring modulators, white noise generators, sample and hold modules, high-pass filters, low-pass filters, band-pass filters, and others. The effect of each module on the signal is different, but the purpose of each is to change the original signal from a plain and uninteresting sound to a rich and intriguing sound.

Digital synthesizers

The Amiga, like many other modern synthesizers, is a digital synthesizer. Instead of producing a simple signal with an oscillator and then enhancing it by passing it through other electronic equipment, a digital synthesizer creates a sound as a digital mathematical model, then turns the model into an audio signal that it sends directly to the amplifier. With digital synthesizers, creating new and unique sounds becomes as much a matter of mathematics as it does electronics.

The Amiga makes its digital models by creating the waveform of a sound in its memory. To understand how it does this, think of how a waveform depicts the quality of a sound. You read a waveform like the one in Figure 7-6 from left to right. The waveform goes up for higher air pressure, down for lower air pressure. This corresponds directly with the motion of the speaker cone: Each time the waveform goes above the horizontal center line, the speaker cone moves out to create higher pressure. Each time the waveform goes below the horizontal center line, the speaker cone moves in to create lower pressure.

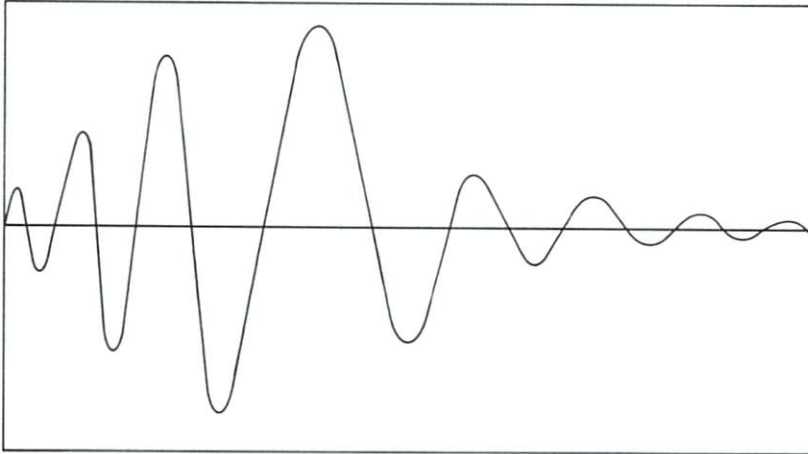


Figure 7-6.

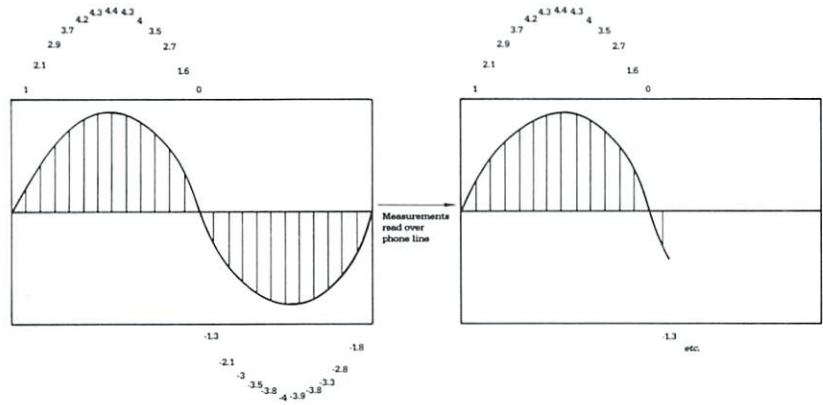
A typical waveform.

As you read the waveform, you can see the entire record of the speaker cone's motion from the beginning to the end of the sound. The more waves there are in the waveform, the faster the cone moves in and out. The higher and lower the waveform goes, the more motion the cone goes through. By sending the waveform of a sound as an amplified audio signal to a speaker, you can create that entire sound.

To store a waveform in the Amiga's memory, it has to be chopped up into a series of numbers. A good way to think of it is to imagine describing a waveform over the phone to another person. You can chop the waveform up horizontally into tiny equal segments, shown in Figure 7-7 (on the next page). By measuring the height of each segment, you can pass the measurements on to the person at the other end of the phone line, who can reconstruct the waveform if you tell him the width of the segments you used.

Figure 7-7.

A waveform chopped into measured segments. The amplitude measurement of each segment is read over the phone line to another person who reconstructs the waveform.



The Amiga creates waveforms in the same way, storing them in its memory as an ordered list of waveform measurements called a waveform table. The measurements use a very small segment size to keep the waveform accurate.

To create complex and interesting waveforms, the Amiga uses mathematical formulas to create a waveform from scratch. These formulas plot each point in the waveform table. Once the waveform table is complete, it's sent to a digital-to-analog converter (usually called a D-to-A converter) that converts the waveform's digital measurements into an unamplified audio signal, which is then amplified and sent to the speakers.

The Amiga has four internal audio channels, each with its own waveform table and D-to-A converter. The outputs of these channels are mixed in pairs and sent out through the left and right audio ports in the back of the Amiga's console, two from the left port and two from the right port. Each audio channel plays notes by looping through its waveform table. You can change the pitch of notes played by a channel by changing the rate the Amiga uses to read through the channel's waveform table: The faster the rate, the higher the pitch.

Although the Amiga has just four audio channels, it's not limited to four voices of music; any single waveform table can store an infinite number of pitches, so four waveform tables can create an infinite number of musical voices. For example, think of the single waveform produced by a microphone in front of a large symphony

orchestra. That waveform has sound information for over a hundred instruments simultaneously playing different pitches. If you play the waveform back over a speaker, you can hear all the different pitches played in the orchestra. A waveform table containing that same orchestral waveform and stored in one of the Amiga's internal audio channels would create the same number of pitches when you played it back.

In practice, getting more than four independent musical voices on the Amiga at once is a complicated process. Using mathematical synthesis to create a multi-pitch waveform table on one audio channel isn't as easy as creating a single-pitch waveform table. Once you create the mathematical model of a multi-pitch waveform table, the pitches in the waveform table are locked together. When the audio channel speeds up or slows down its playback rate to raise or lower the pitch, all the pitches are raised or lowered together, so they aren't actually independent musical voices. To change pitches independently within a multi-pitch audio channel, the Amiga has to calculate a new waveform table for each individual pitch change. This requires some tricky programming, but it's not impossible, so look for software that can play eight or even sixteen independent voices of music on the Amiga.

When you use your Amiga as a synthesizer, you don't have to mathematically calculate your own waveform tables (although you can, if you're an advanced programmer). Instead, you can use a variety of methods offered by different Amiga synthesizer programs. For example, some music programs use on-screen controls that look very much like the controls of an analog synthesizer. Other programs may use menus to change various aspects of sound, or they may let you draw waveforms directly on the screen. No matter what controls the program provides, they all change the mathematical formulas that create the waveform table inside the Amiga.

Sampled sounds

The Amiga and some other digital synthesizers can create sounds without synthesizing them. Instead, they record the sounds from the outside world, storing them in a waveform table as a series of waveform measurements. These recorded sounds are called sampled sounds—the measurement of each segment of the waveform is called a sample, and the process of making those measurements and storing the waveform as a series of numbers is called sampling.

To create a sampled sound, you need to add a microphone and a sampler to the Amiga. When the microphone picks up a sound, it

sends the waveform of the sound as an unamplified audio signal to the sampler. The sampler chops the waveform up into approximately 20,000 segments per second (the number of samples varies with the sampling rate), takes a reading of the amplitude of each segment, and sends each sample to the Amiga in the order it was taken. This stream of samples is stored in the Amiga's memory as a waveform table, and can be played back through an audio channel's D-to-A converter just as a synthesized waveform is played back.

Sampled sounds are usually very rich, since they're recorded directly from a very complex acoustic source. They do have a drawback, though; they need a lot of RAM for storage, because each individual sample is stored in memory, unlike a synthesized waveform, which is stored as a formula. When you consider that each sample of a sound needs one byte to store it in memory, and that a sampler samples 20,000 segments per second, a sampled sound of two seconds needs 40,000 bytes. This adds up quickly if you use many sampled sounds at once in the Amiga. Synthesized sounds, on the other hand, require a very small amount of memory. Instead of storing the entire waveform table in memory as a series of individual numbers, the Amiga stores only the mathematical formula needed to create a synthesized sound. When you want to play a synthesized sound, the Amiga recalls the formula and creates the waveform from it.

Sampled sounds are inflexible if played back normally; the waveform plays back at the same pitch, loudness, timbre, and length as the original sound—it's just a digitized recording of the sound. Fortunately, the Amiga can alter the playback of a sampled sound to use it musically. By increasing and decreasing the values of the samples, it can change the loudness of the sound. By sending the samples to the D-to-A converter at a slower or faster rate, it can change the frequency of the sound.

To change the duration of the sampled sound, the Amiga must repeat part of it over and over. It's important that the repeated section sound smooth, so somebody has to listen to the sound produced by a sampled-sound waveform to find the best section to repeat (called the sustain loop). If the sustain loop is too near the beginning of the waveform, you might hear the attack of the sound over and over again—the "pick" of a guitar string or the blurble of a trumpet note—a very distracting effect. If the sustain loop is set too near the end of the waveform, the sound may have died away to nothing, so repeating it does no good.

Obviously, setting a sustain loop is a matter of judgment. The sampled sounds you get with a program like Deluxe Music have

the sustain loop already picked out and recorded with the sound on disk. To create your own sampled sounds, you'll need extra software to go with the sampler and the microphone you use to capture the sampled sounds. Once you have the sounds, you'll have to use the software to look at their waveforms on the screen and adjust the beginning and end of the sustain loop until you find the best sound.

SEQUENCERS

A musical sequencer is a device that controls and plays a musical instrument. For example, a player piano uses a roll of paper containing holes to play the piano keys. Each hole on the roll corresponds to one key pressed on the piano keyboard. An electronic music sequencer works on similar principles; it stores notes electronically and uses them to play music on a synthesizer. Early sequencers were simple affairs—electromechanical devices that stored a sequence of pitches and nothing more. They played perhaps 20 to 50 pitches on the synthesizer, all the same duration, and then repeated them over and over again until the sequencer was stopped.

Today's sequencers are much different. They use computers, and they're no longer restricted to a small set of pitches. They can store hundreds of thousands of pitches. They can also control durations of notes and regulate other musical factors such as volume, instrumentation, and timbre.

Sequencer programs on the Amiga can be very sophisticated. You can enter notes into them using a traditional music staff or a notation system designed especially for the program. The sequencer software stores the notes as a sequence of events with precise timing. The events can be items like the start of a note, the end of a note, an instrument change, a volume change, and other musical factors. When the sequencer plays back its score, it sends these events to the Amiga's built-in synthesizer as control signals. These control signals act like a phantom musician, starting and stopping notes, and twiddling synthesizer "dials" to change the tonal quality of the sounds.

The advantage of using a sequencer over playing a keyboard is that the sequencer can perform music that no one human musician could create. It can control more voices, play faster and more complex music, and transpose to any key or change to any tempo with a single command. The disadvantage is that no sequencer is spontaneous. It has to be programmed beforehand, and can't react to the mood around it, changing its performance to respond to the audience.

SYNTHESIZING SPEECH

If you listen closely to yourself speaking, you can hear the small elements of speech—changes in timbre when you change vowels and consonants, and changes in pitch and volume as you speak with inflection, accenting different syllables. By synthesizing a waveform that includes all these quick and very subtle changes in timbre, pitch, and volume, it's possible to make a synthesizer speak.

The Amiga has a device in its system software that makes speech synthesis easy. It takes text in strings of letters from sources such as the keyboard or memory, and turns them into waveforms that create speech instead of music when they're converted into sound on an audio channel. The text to be translated into speech is spelled out phonetically, using a special computer phonetic spelling; that is, using a spelling like "LAH5NCHWAE2GIN" for a word like "lunchwagon." (The Amiga's phonetic spellings are discussed in Chapter 9, along with other details of speech synthesis.)

Phonetic spelling gives you detailed control of word pronunciation and syllable inflection, but it takes some time to translate normal English words into the correct phonetic spelling. To make things easier, the Amiga's system software also contains devices that translate strings of English words into strings of phonetic spellings to be turned into synthesized speech. When you consider how many different pronunciations exist for some letters in English (take the "o" in "women" for example), it's quite a feat to decide which pronunciation should be used for a letter. The translation devices in the Amiga's system software do it quite well.

The speech device can also alter the quality of the Amiga's speaking voice. It can speak in a high or low voice to sound male or female, and it can use different pitches to sing lyrics to a song. It can speak with a lot of inflection or in a monotone. It can also change the volume and the speed of the speech.

RECORDING ELECTRONIC MUSIC

Playing music on the Amiga is great fun, but most people won't get a chance to hear your creations unless you drag your Amiga out in the streets or you invite large numbers of people over to stand in your computer room. The best solution is to record your music on tape. Anyone with a sound system is likely to own a cassette player, so if you gave them a tape they can play back your

music. You can copy the tape and send it off to faraway relatives and recording executives, and you can play it back to yourself in your car tape deck as you drive.

Recording on tape has additional advantages: You can mix in sounds from sources outside your Amiga. For example, if you enter the accompaniment to "Ode to the Wombat" on the Amiga, you can play it back while singing the words into a microphone, and record the whole performance for posterity. You can also blend the left and right audio channels of the Amiga together to image their sounds, and use some tricks to record your Amiga playing with your Amiga. All you need is the right equipment.

TAPE RECORDERS

The most important piece of equipment you'll need is a tape recorder. The most common type is the cassette recorder, but you can also use a reel-to-reel recorder or a videocassette recorder (hi-fi VCRs have impressive audio recording capabilities). Use a cassette recorder if you want a tape medium that's easy to use and easy for most people to play back. Consider a reel-to-reel recorder if you plan to edit your recordings—reel-to-reel recordings are much easier to edit than cassette or video tapes.

Most tape recorders are stereo recorders: They use two discrete tracks on the tape. Each track stores a different audio signal that plays back to create the two signals needed for stereo sound reproduction. Some tape recorders (usually reel-to-reel recorders) have more than two tracks, ranging from 4 to 32 discrete tracks. Each individual track can record a separate signal. Multi-track recorders (the type used in recording studios) are used to record one musical line per track so the volume and tonal quality of each line can be adjusted individually when you play it back. Recording one track at a time allows a single musician to create a tape of himself playing different instruments simultaneously, so he ends up performing an entire song by himself.

MIXERS

To feed many different audio signals into a tape recorder, you need a mixer. Mixers can adjust the strength of each individual audio signal to give you a good balance between them. For example, if you plan to record using a microphone and your Amiga, you may find the microphone signal to be weaker than the signal from the Amiga. A mixer will adjust for the differences in signal strengths and can also help to create imaging effects in a stereo recording. A mixer can take the two discrete sounds coming from the left and right audio jacks of the Amiga and blend them across the left and right channels of the recorder. A small knob called a pan pot lets you position the apparent location of the sounds from the Amiga.

SIMPLE RESULTS

If you just want to create a simple recording of your Amiga, you don't need fancy equipment. One cassette recorder with two line-level input jacks will work. (A line-level signal is an unamplified audio signal.) The audio jacks on the back of the Amiga produce a line-level output, which can be directly connected to the input jacks of a cassette deck using two audio cables. All you need to do is put a cassette in the recorder and start recording.

MIDI

Although the Amiga by itself is capable of impressive music, it can be turned into a much more powerful system by connecting it to outside synthesizers using MIDI. MIDI stands for Musical Instrument Digital Interface. It's not a piece of equipment itself; it's a standard that regulates the type of cables and the information format that are used to send musical information back and forth between synthesizers, or between synthesizers and a computer.

MIDI evolved a few years ago as a way to simplify synthesizer connections. You probably have memories of a rock concert where the keyboard player would jump back and forth between racks of synthesizers to play a riff on the one synthesizer that had just the sound he wanted. When synthesizers are connected together with MIDI cables, a musician can use the keyboard of any connected synthesizer to play the sounds of any other connected synthesizer. It's possible for a musician to set up one synthesizer as the master synthesizer, using its keyboard to play the sounds of the other synthesizers as well as its own sounds. The musician can choose which synthesizer (or synthesizers) he wants to play with the master keyboard, and so saves himself a lot of jumping around.

MIDI is flexible enough to allow many other possibilities. For example, you can include a computer in the MIDI network. Since MIDI carries information for all the keystrokes, pitch wheel bends, instrument changes, and other actions that a performer might take as he plays, the computer can create a record of all those activities, recording not sounds, but performance events. It can re-create the music later by sending the recorded MIDI events back out over the

MIDI connection to attached synthesizers. The computer used in this type of arrangement is called a MIDI recorder, and is very useful, since the MIDI events stored in memory are easy to modify.

THE MIDI PHYSICAL STANDARD

MIDI has become a music industry standard. Almost all synthesizer manufacturers include at least one MIDI port on their synthesizers so you can connect them to other MIDI-equipped machines. Standardization makes it simple to connect MIDI equipment together—you use a MIDI cable with 5-pin DIN plugs (a standard type of audio plug) on each end, and plug one end of the cable into the MIDI port on one machine and the other into the MIDI port on a second machine.

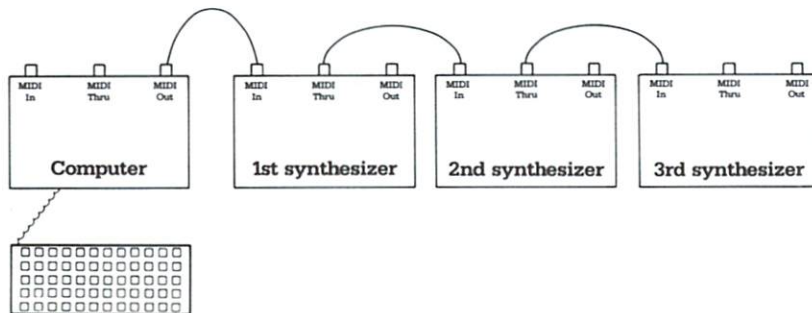
There are three different kinds of MIDI ports used on MIDI equipment: MIDI In, MIDI Out, and MIDI Thru. The MIDI In port receives signals from another MIDI device. The MIDI Out port sends out signals from that device to another MIDI device. MIDI Out usually just sends signals that originate from that device (like keystrokes, etc.), but on some equipment it can also mix in the signals the device receives through the MIDI In port, and pass both signals through the MIDI Out port. The MIDI Thru port simply passes on signals that come in through the MIDI In port without changing them.

You can use the three types of MIDI ports to connect MIDI equipment in different configurations. For example, you can connect a computer to a synthesizer by connecting the computer's MIDI Out port to the synthesizer's MIDI In port, and by connecting the computer's MIDI In port to the synthesizer's MIDI Out port. Using this configuration, the computer can read all the MIDI signals that the synthesizer sends, and the synthesizer can read all the MIDI signals that the computer sends.

If you have a computer and several synthesizers, you can set up the computer as a controller for the synthesizers. To do this, you connect the computer's MIDI Out port to the MIDI In port of the first synthesizer. Next, connect the first synthesizer's MIDI Thru port to the MIDI In port of the second synthesizer, then connect the second synthesizer's MIDI Thru port to the MIDI In port of the third synthesizer. You can also connect additional synthesizers by connecting each new synthesizer's MIDI In port with the MIDI Thru port of the last connected synthesizer. With this configuration, each synthesizer is only capable of receiving (via its MIDI In port) MIDI messages sent out by the computer, or passing along MIDI messages to the next synthesizer (via its MIDI Thru port). An example of this configuration is shown in Figure 7-8 (on the next page).

Figure 7-8.

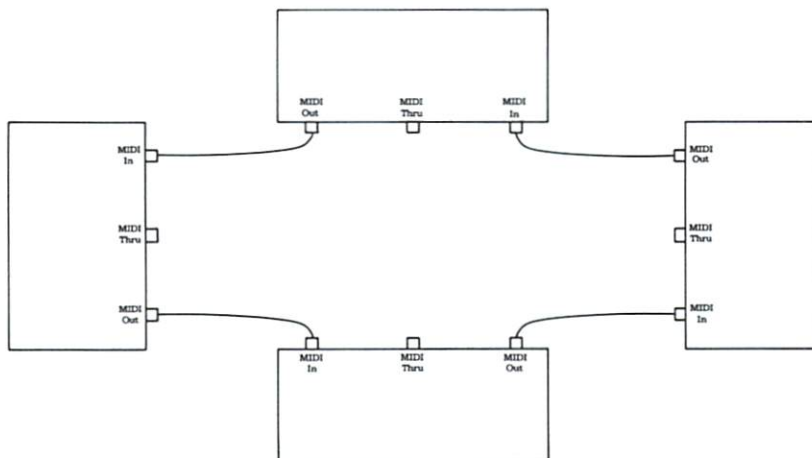
Connecting a computer to MIDI devices to use it as a controller.



Another configuration that allows each MIDI device to exchange information with any other MIDI device is a MIDI "ring," shown in Figure 7-9. To create a ring, all the MIDI devices must be equipped to mix the MIDI In signals they receive with the signals they create, and pass both signals out through the MIDI Out port. This configuration allows any device in the ring to exchange MIDI messages with any other device in the ring.

Figure 7-9.

Creating a MIDI ring to allow MIDI message interchange between devices.



To use the Amiga as a MIDI machine, you need to add a MIDI adaptor to the serial port on the back of the Amiga's console. The MIDI adaptor has the three different MIDI ports, and makes the Amiga serial signal compatible with the MIDI standard signal.

MIDI MESSAGES

The signals that pass from machine to machine over the MIDI cables are similar to the signals that pass from computer to computer when the computers are connected together with a modem. Modem communication normally uses a type of code called ASCII (American Standard Code for Information Interchange) that uses a set of code numbers to represent letters of the alphabet, numerals, punctuation marks, as well as computer information such as carriage returns, line feeds, and other control characters.

MIDI also uses a set of code numbers, but instead of representing letters and numbers, each one stands for different occurrences on a synthesizer. These MIDI code numbers are strung together in groups called messages. Typical MIDI messages communicate events like note-on (that a keyboard key was pressed, what pitch it was, and how hard the key was pressed), note-off (that a key was released and what key it was), and that a pitch wheel was moved (how far the pitch wheel, a device that bends pitch up or down, was moved).

MIDI messages are sent over the cables in 16 different software-controlled channels. Each channel carries messages for one MIDI machine (you specify which machine uses which channel), so it's possible to connect 16 different MIDI machines together on the same MIDI network with each machine sending and receiving MIDI messages on its own channel. Each MIDI channel can transmit all the events happening on its synthesizer, so each channel can transmit a full performance—chords, fancy fingerwork, instrument changes, and all. A central computer controlling external synthesizers can use the separate channels to send messages to and record events from individual synthesizers.

Now you know how sound is created, how you hear it, and how it can be electronically produced. You've learned how sound synthesis works, and how the Amiga can synthesize its own sounds and speech and re-create sampled sounds. You've also been introduced to MIDI, and learned how to attach external synthesizers to the Amiga. With the information you gained in this chapter, you're now well equipped to go on to the next two sound chapters and start making your own sounds, music, and computer-generated speech.



**CHAPTER EIGHT
AMIGA MUSIC
TOOLS**

Today's equivalent of Johann Sebastian Bach ruining his eyesight by poring over scores in candlelight is the modern musician staring at his computer monitor in the wee hours of the morning. Although the tools have changed, composition is still a combination of inspiration, structuring, recording, and struggling to bring the music to performance. The Amiga can't help you with the inspiration and creative process, but it can relieve a lot of the drudgery involved in writing your music and getting it played. In this chapter, you'll learn some Amiga music techniques that will enable you to spend more time listening to your music and less time staring at your monitor.

A wide variety of music programs has been written for the Amiga. Some programs record the notes of your music in the Amiga's memory and play them back at your request; others let you play on an attached keyboard or on the Amiga's keyboard, using the Amiga as a musical instrument. There are also programs that let you create and modify your own Amiga synthesized instruments. The program featured in this chapter, Electronic Arts' Deluxe Music, concentrates on storing notes and playing them back. Deluxe Music lets you enter scores with traditional music notation and play back your scores using the Amiga's built-in synthesizer or an attached MIDI synthesizer.

This chapter teaches you advanced Deluxe Music techniques. You'll learn special methods for entering tempo and instrument changes, using 64th notes, and working with MIDI instruments. You'll also find some tips and hints that make score entry easy. Later sections in the chapter show you how to record your musical creations on tape, and take you on a tour through other music software and hardware available for the Amiga.

MASTERING DELUXE MUSIC

Deluxe Music is a sophisticated program that you can use to enter, edit, play back, and print music. It uses traditional music notation—staves with notes—to display musical scores, but offers a nontraditional note-entry method that makes it quite easy to enter musical scores. You can either place notes of different durations directly on the staff using the mouse, or, if you're more familiar with a music keyboard, you can enter pitches by clicking

the mouse pointer on the keys of a music keyboard Deluxe Music displays on the bottom of the screen. If you have a MIDI keyboard connected to the Amiga, you can also enter pitches by pressing its keyboard keys.

Deluxe Music's score-entry system is quite powerful. It provides up to eight staves of music on the screen at one time, and allows you to choose treble, bass, alto, or tenor clefs for each of the staves. You can enter notes and rests that range from a whole note to a 32nd note, dot any note, and use the notes to create triplets or quintuplets if you like. You can tie and slur groups of notes; put in crescendos, diminuendos, and dynamic markings ranging from *ppp* (very soft) to *fff* (very loud); and change time and key signatures within the score.

Deluxe Music has powerful editing features that let you cut, copy, and paste music much like a word processor works with text. You can transpose groups of notes up and down in pitch. Deluxe Music's features let you make the score more visually appealing, both on the screen and on the printed copy. You can set the direction of note stems and beam groups of 8th, 16th, and 32nd notes together for clarity. (Beaming individual notes together, as shown in Figure 8-1 on page 199, makes them easier for a musician to read.) You can reposition the notes, rests, and staves for maximum clarity, and add text to the score for lyrics, tempo markings, or directions to musicians. Deluxe Music has a special built-in music font that contains music symbols like violin up and down bow markings, trill markings, and other characters useful in a music score. When the score looks just the way you want it, you can print it out on an attached printer.

Deluxe Music plays back your scores using sampled sound instruments that it loads from disk into the Amiga's memory. Once the instruments are in memory, Deluxe Music plays them using the Amiga's four internal audio channels. You can orchestrate your score by making instrument changes at different locations in the score. Using only the Amiga's four audio channels limits Deluxe Music to a maximum of four notes played at once, but you can add extra voices by attaching an external MIDI synthesizer. Deluxe Music will use the external synthesizer's instruments in addition to the Amiga's instruments (a very handy feature, considering that Deluxe Music lets you enter a large number of simultaneous notes on up to eight staves).

STRETCHING DELUXE MUSIC'S CAPABILITIES

Many people use Deluxe Music to enter scores directly from sheet music, copying the music note by note so they can hear the Amiga play it back. It's a simple task to enter just the notes and rests you see on the page. Unfortunately, music entered like this often sounds mechanical or bland. Notes and rests are merely the framework of music—the real soul and expression come from the musician's flexibility and spontaneity while playing each of those notes back.

Deluxe Music offers several features that help to give your scores style in playback: You can use crescendo and diminuendo along with dynamic markings to make your music rise and fall expressively in volume. You can slur groups of notes together so they play smoothly from note to note, or you can make notes staccato so each one jumps out at you as an individual note. These features are available on a note-by-note basis: You can start the effect at any note in the score and end it on any other note in the score.

Deluxe Music has other features to add pizzazz to your playing style. For example, you can start playing a voice in the score using one instrument and then change to another instrument to give it a different timbre—sudden timbre changes can really perk up a listener's ears. You can also change the tempo of the score playback in the middle of the score; suddenly speeding up or slowing down playback can also have a dramatic impact on a listener. Although these two playing-style features are very useful, Deluxe Music doesn't let you apply them on a note-by-note basis. You can only start them at the beginning or end of a measure; you can't change tempo or instruments in the middle of a measure.

Most music uses measures only as a timing reference, so the bar lines in a score don't always fall at the precise point where you want to change an instrument or the tempo. In fact, if you want to use *accelerando* (where the tempo steadily increases over a group of notes) or *ritardando* (where the tempo steadily decreases over a group of notes), Deluxe Music's ability to jump to a new tempo at every bar line won't give you the effect you want if you use ordinary measures.

Fortunately, Deluxe Music lets you create measures of almost any size, so you can get around the measure limitations of tempo and instrument changes to stretch Deluxe Music's capabilities. By using some of the **Measures** menu commands like **Insert Measure**, **Split Measure**, and **Join Measure**, and then setting a different time signature for each measure, you can rearrange the bar lines in a score to fall exactly where you want them to fall for changes. The next section shows you how to take advantage of this feature.

Changing Tempos: Accelerandos and Ritardandos



Figure 8-1.

An example of ritardando in two measures of music.

Figure 8-1 shows the last two measures of a piece of music. To make the ending sound substantial, there is a ritardando (marked by the "rit" sign and a dotted line) that slows down the last eight 16th notes going into the final whole note. To enter this short phrase with the ritardando in Deluxe Music, follow these instructions:

1. Use the **New Score** command from the **File** menu to create a new score.
2. Use the **Set Time Signature...** command from the **Measures** menu to make sure the time signature of the score is set to 4/4 time.
3. Enter the notes as you see them in Figure 8-1: sixteen 16th notes in the first measure and a whole note in the second (last) measure.
4. Choose **Score Setup** from the **Window** menu and make sure the **Beats per Min** slider is set to 90.
5. Try playing the score. It should play back at 90 beats per minute with no ritardando.
6. Use the editing arrow to put a note-entry cursor (a blinking vertical line) between the eighth and ninth 16th notes of the first measure.

7. Use the **Split Measure** command from the **Measures** menu to insert a new bar line at the location of the note-entry cursor, splitting the first measure into two new measures, each with eight 16th notes.
8. Use the editing arrow and the **Split Measure** command to split the second new measure into four measures, each measure with two 16th notes.

You should now have six measures in the score: the first measure with eight 16th notes, measures two through five with two 16th notes, and measure six with a whole note. To see all six measures at once, use the editing arrow to drag the right boundary of each measure to the left until all measures are visible on the screen.

If you play the score now, you'll hear the 16th notes play back with long pauses where you inserted the new bar lines. This happens because the measures are all 4/4 measures, and aren't filled up. To tighten the rhythmic slack, you must change the time signatures of the new measures:

1. Change the first measure to 2/4 time by clicking in the measure with the editing arrow, then using the **Set Time Signature...** command in the **Measures** menu to change the time signature to 2/4.
Deluxe Music automatically changes the measure you click in and all the measures after it to the new time signature, so when you changed the first measure, all the following measures also changed to 2/4 time. However, you want to use 2/4 time for the first measure only, so you'll have to change the time again for the following measures.
2. Change the second, third, fourth, and fifth measures to 1/8 time by clicking in the second measure with the editing arrow and using the **Set Time Signature...** command again.
3. Use the same method to set the sixth measure's time signature back to 4/4. Now each measure's time signature is set so the measure's contents fill it up completely.

If you play back the score now, it sounds just like it did when you first entered it. If it seems like you just did a lot of work for nothing, keep in mind that you now have a lot of handy bar lines sprinkled through the notes, so you're ready to add a ritardando:

1. Play back the score at different tempos by moving the **Beats per Min** slider in the **Score Setup** window to find a slow tempo that you want the ritardando to reach by the last measure. For this example, 90 beats per minute works well for the initial tempo, slowing down to 56 beats per minute by the final whole note.
2. Set the tempo in the first measure to 90 beats per minute by selecting the first measure, making sure the **Beats per Min** slider in the **Score Setup** window is set to 90, and then using the **Set Tempo** command in the **Measures** menu. The tempo marking will appear at the beginning of the measure.
3. Set the tempo in the last measure to 56 beats per minute by selecting the last measure, setting the slider to 56 in the **Score Setup** window, and then using the **Set Tempo** command.
4. To create a ritardando, change the tempos of the four middle measures in increments that go from 90 to 56. Values of 82, 74, 67, and 61 work well. The measure tempos in the final score should read: 90, 82, 74, 67, 61, and 56.

The final score should look like Figure 8-2. If you play back the score now, you'll hear the ritardando as the tempo gets progressively slower toward the end of the score.

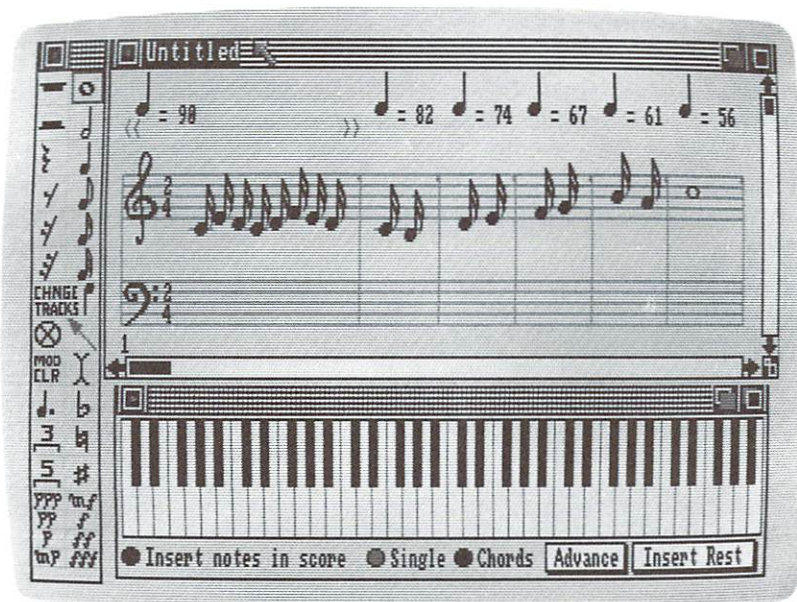


Figure 8-2.

Split measures and changing tempos produce a ritardando in Deluxe Music.

If you use this method with two or more staves, you should note that whenever you split a measure in one staff, Deluxe Music splits the corresponding measures in any other staves of your score. If you split the third measure of the first staff, for example, the third measure in the other staves will be split as well. To compensate for this, you'll have to break up the notes in the other staves to match the measures you create for your ritardando. As an example, if you had a second staff accompanying the music in the last example, and the second staff had a whole note in the first measure, you'd have to break up the whole note into a single half note and four eighth notes to fit in the new measures, then tie them together to make them sound like one note.

Fast Instrument Changes

If you use Deluxe Music with just the Amiga's internal voices, you're limited to four different instruments playing at a time. While you can get a lot of color with four instruments, there are times when it would be nice to have even more instruments. You can create the illusion of more than four voices playing at a time if you set up more than four voices in the score and switch quickly back and forth between them.

As an example, think of a rhythm section in a band that includes a bass guitar, a snare drum, a high-hat cymbal, and a bass drum. When they play music together, the bass guitar plays the down beats while the percussion instruments alternate to fill in the back beats. If you enter a score like this in Deluxe Music where none of these instruments play together at the same time, then they will use only one of the Amiga's internal voices. This effectively gives you four instruments for the price of one voice. In Figure 8-3, you can see a rhythmic pattern that has been divided between five different instruments this way.

Figure 8-3.

A rhythmic pattern is divided between many alternating instruments.

The image shows a musical score for five instruments: Snare Drum, Bass Drum, Cymbal, Clave, and Tom Drum. The score is written in 4/4 time and consists of two measures. The Snare Drum staff has a rhythmic pattern of quarter notes in the first measure and quarter notes in the second measure. The Bass Drum staff has a rhythmic pattern of quarter notes in the first measure and quarter notes in the second measure. The Cymbal staff has a rhythmic pattern of quarter notes in the first measure and quarter notes in the second measure. The Clave staff has a rhythmic pattern of quarter notes in the first measure and quarter notes in the second measure. The Tom Drum staff has a rhythmic pattern of quarter notes in the first measure and quarter notes in the second measure.

To try the two measures in Figure 8-3 in a Deluxe Music score, follow these instructions:

1. Create a new score. Open the **Score Setup** window and add three staves to the score. This will give you a total of five staves for the five different instruments in the music. Set the score to C major in 4/4 time.
2. Load from disk the instruments for the score. You need five instruments: a bass drum, a snare drum, a cymbal, a clave, and a tom drum. (If you don't have these instruments, you can approximate with five other instruments you think might sound similar.) To load them, choose the **Load Instrument...** command from the **Sounds** menu five times, selecting a different instrument each time so the five instruments appear at the top of the **Sounds** menu.
3. To assign the snare drum to the first staff, select the first measure of the staff, choose the snare drum from the **Sounds** menu, then choose **Set Instrument** from the **Measures** menu. The name of the instrument should appear just above the beginning of the first measure.
4. Use the same method to assign the bass drum to the first measure of the second staff, the cymbal to the first measure of the third staff, the clave to the first measure of the fourth staff, and the tom drum to the first measure of the fifth staff.
5. To enter the snare drum notes, select the first measure of the snare drum staff, then enter all the snare drum notes, filling in with rests where there are notes in the music that will be played by other instruments.
6. Use the same technique to enter bass drum, cymbal, clave, and tom drum notes in their respective staves.

When you play back your score, you should hear a full rhythm section that uses only one Amiga voice for each note. If you want to add more voices to the score, you can add three more staves and enter a part using a different instrument in each of the staves. For example, you could have a lead guitar line in the sixth staff, a sax line in the seventh staff, and a piano line in the eighth staff.

As you enter notes for these new voices (or for that matter any score with four voices), be careful where you use chords. You can use up the Amiga's voices quickly, and this may leave some notes unplayed. For example, if you put a three-note chord in the piano voice at the same moment that the other three voices are playing notes, you have a total of six simultaneous notes. Since Deluxe Music can play only four of those notes on the Amiga, two of the notes won't be played.

To make sure important instruments are played in case of too many simultaneous notes in a score, you can use a louder dynamic marking for the staff you want to be heard. Deluxe Music will always play the notes in a staff set with a higher dynamic before it plays notes in a staff set to a lower dynamic. For example, if you have five different staves playing notes at the same time, and the staves are set at *pp*, *mp*, *mf*, *f*, and *ff*, then Deluxe Music will play only the notes on the four loudest staves; it won't play the notes in the staff set at *pp*. By setting one staff slightly louder than the others, at *f*, for instance, while all the others are set at *mf*, you can ensure that staff will always be heard, even if it isn't significantly louder than the other staves.

Handling 64th notes

Deluxe Music gives you a wide range of note lengths: You have six basic notes that you can combine with dots and triplet and quintuplet markings to come up with a total of 36 different notes. (Of course, some of these, like quintuplet dotted half notes, are just a wee bit esoteric.) With all these available notes, it is still possible to exceed Deluxe Music's note values: One of these days you'll be copying a score into Deluxe Music, and 145 measures into the score you'll come to a screeching halt at a quick run of 64th notes. Then what will you do?

There is a simple solution. It centers around the fact that note lengths are purely relative; they have no fixed duration. For example, if you enter a score with quarter notes and eighth notes and play the score back at a tempo of 60 beats per minute, the quarter notes will each last exactly one second, and the eighth notes will each last exactly one half second. If you change the tempo to 120 beats per minute, though, each quarter note will last exactly one half second, and each eighth note will last exactly one

fourth second. The note durations are entirely flexible, and depend on the tempo at which you play the music. The relative lengths of the notes are fixed, however: A quarter note is always twice as long as an eighth note, a half note is always twice as long as a quarter note, and so on.

As you enter music using different note lengths, you aren't actually entering a series of time durations, you're entering a series of relative lengths—this note is twice as long as this one, this note is a third as long as this one—and so on throughout the whole score. It doesn't actually matter what notes you use, the only thing that matters is how long the notes are in relation to each other.

When you want to copy a run of 64th notes from printed music, you won't be able to find notes in Deluxe Music's note palette that are half as long as the 32nd notes in the score. You can make the proportions work out, however, by making all the other notes in the score twice as long and then entering the run as 32nd notes. Try this example:

1. Create a new score and set it to 6/8 time in C major.
2. Enter the notes of the music in Figure 8-4 until you reach the 64th notes at the end of the second measure.



Figure 8-4.

Music with a run of 64th notes.

3. To accommodate the 64th notes, you must double the length of all the other notes in the score. Choose **Select All** from the **Edit** menu to select the entire score. Choose **Double Time** from the **Notes** menu. This doubles the length of all the notes in the score (eighth notes become quarter notes, and so forth), but the notes in the last half of each measure are shaded, because the **Double Time** command has no effect on the length of the measures themselves. Deluxe Music shades these notes to indicate that they no longer fit within the time signature in effect for that measure.

4. To accommodate the new note lengths, you'll have to double the time signature of the measures. Select the first measure of the score, then reset the time signature to 6/4 time using the **Set Time Signature...** command in the **Measures** menu.
5. Enter the 64th notes as 32nd notes at the end of the last measure. If you play back the score now, all the notes will have the same relative length to each other, but the note values themselves have been doubled, so the playback tempo is half as fast as it was previously.
6. Double the playback tempo of the score by choosing **Score Setup** from the **Window** menu and using the **Beats per Min** slider to double the beats per minute of the current tempo setting. When you're finished, you can apply the new tempo to the first measure of the score by selecting the measure and choosing **Set Tempo** from the **Measures** menu.

Now when you play back the score, it sounds just like it would if you had entered it the way it was written and played it at the original tempo. You have entered the score using 32nd notes to produce the rhythm that was originally written in 64th notes.

If you go on to enter more notes in a score in which you've doubled the tempo, be sure to double all the note and rest lengths as you enter them from the printed score. As a shortcut, you can enter the notes and rests as they are written, then select them and double them with the **Double Time** command.

Whenever you use this method while entering scores, be sure to double the time signature of all the measures in the score. If you have time-signature changes within the score, doubling just the first time signature won't affect the measures with the time-signature changes, so you have to double each changed measure individually. To double a time signature, it's easiest just to halve the number at the bottom of the time signature. For example, 4/4 time doubled is 4/2, 3/8 doubled is 3/4. If you can't halve the bottom number, then you can double the top number. For example, 3/1 doubled is 6/1.

USING DELUXE MUSIC WITH MIDI

One handy feature of Deluxe Music is that you can use it with an external MIDI synthesizer to expand the number of instruments that can play at one time in your scores. To set up a score to play using a MIDI synthesizer, you use the **Set Instrument** command in the **Measures** menu very much like you would to set an Amiga instrument.

To use one or several MIDI synthesizers with Deluxe Music, you must first connect them to the Amiga with a MIDI adaptor. Connect the first (or only) synthesizer by running a MIDI cable from its MIDI Out port to the MIDI adaptor's MIDI In port, and running another cable from the adaptor's MIDI Out port to the synthesizer's MIDI In port. To connect additional synthesizers, run a cable from the first synthesizer's MIDI Thru port to the second synthesizer's MIDI In port, another cable from the second synthesizer's MIDI Thru port to the third synthesizer's MIDI In port, and so on until you have all the synthesizers connected. Examples of these connections, along with a more detailed explanation of MIDI, can be found at the end of Chapter 7.

You must set up each synthesizer so it receives MIDI messages on its own MIDI channel (unless you use just one synthesizer). You'll have to do some reading through the synthesizers' users manuals to see how to set the incoming MIDI channel. If you can't set a synthesizer's incoming MIDI channel, chances are the synthesizer receives messages on MIDI channel 1.

Once you have the connections made, you can turn on the synthesizers and use them with Deluxe Music. If you want to play part of the score through the synthesizer, first set up MIDI using commands in the **Sounds** menu:

1. Choose the **MIDI Active** command to load the MIDI driver from the Deluxe Music disk into the Amiga's memory.
2. Use the **MIDI Channel...** command to choose any one of the 16 MIDI channels that you want to use. Since you have the connected synthesizers all set to different channels, you can pick which synthesizer you want to play by choosing its channel.
3. Choose the **MIDI Setup...** command to open a requester where you can choose a preset number. Presets on a synthesizer are the different built-in instrumental sounds the synthesizer can play. You've probably played with an electric organ where you could push buttons to get "Trumpet," "Flute," and other sounds—these are examples of presets. MIDI synthesizers usually number their presets so you can choose them by remote control using a preset number. Again, you'll have to delve into the contents of your synthesizer's users manual to find out what these numbers are. Once you know them, you can choose a preset number in this requester using the **MIDI Preset Number** slider to get the sound you want from the synthesizer. Choose **OK** to close the requester.

Now that the MIDI connection is set up for the specific sound you want on a specific synthesizer, you put the setting in your score the same way you choose an instrument:

1. Select the measure where you want the synthesizer to start playing.
2. Choose the **Set Instrument** command from the **Measures** menu. The MIDI setting that you last set up appears above the measure.

You can change synthesizers and synthesizer presets at any point in a score by first changing the MIDI setting in the **Sounds** menu and then using the **Set Instrument** command just as you do to change an instrument in a score.

SOME DELUXE MUSIC TIPS

Here are a few tips you can try to make entering scores in Deluxe Music much easier.

Mark your printed scores

If you enter music from printed scores, invest in a set of colored pencils. Before you start using Deluxe Music, go through the score first and number each measure if they're not already numbered. Go through again, and mark which notes you want to play with different instruments. If you use a different colored pencil for each instrument, it will be easier to see where each instrument is playing, where notes get mixed together on the page, and where each new instrument comes in.

As you enter the score in Deluxe Music, you can match the measure numbers in the Deluxe Music score to the measure numbers on the printed page to make sure you're entering music from the right spot. Of course, if you split measures for ritardandos and other effects, you may find your measure numbers off a bit. You can avoid that by marking the measure splits on your printed score before you number the measures. You can use the instrument markings you made in the printed score to help you enter notes in the correct Deluxe Music staff. Assign one color to each staff you want to use, then enter only the notes you marked in that color to the staff.

Use keyboard shortcuts

Since the note palette is displayed on the screen in plain sight, it's easy to forget that you often don't need to use it. Deluxe Music lets you use keyboard shortcuts instead to choose the length of note you want to enter in a score. Read about the shortcuts in the manual and use them! They make life much easier. As musical inspiration is flowing through your head and you're getting into the swing of entering a melody on a staff, it's much easier to quickly press a function key with your non-mouse hand to choose a new note value than it is to move the pointer all the way to the side of the screen, select a new note, and then try to find your staff location and inspiration a second time.

MAKING A CASSETTE RECORDING USING DELUXE MUSIC

To make a cassette recording of your music, all you have to do is connect your Amiga to the cassette recorder, start the recorder, and then start the music playing. Here are the details you'll need to make a recording using Deluxe Music.

The easiest cable to use for connecting the Amiga to the recorder is a stereo patch cord, which has two phono plugs on each end, and carries two separate signals from the Amiga to the cassette recorder. To connect the cable, plug the two plugs on one end of the patch cable into the two audio jacks labeled with the speaker icons on the back of the Amiga's console. Note which colored plug goes into which audio jack; as you face the back of the Amiga, the jack on your right is the left stereo channel, the jack on the left is the right stereo channel. (If this seems topsy-turvy, consider that when you face the Amiga console from the front, the left port is then on the left, and the right port is on the right.)

On the back of the cassette recorder are two jacks probably labeled either REC IN or LINE IN, each with either an R for right or an L for left. Plug the appropriate colored plug into these two jacks so you match the left and right Amiga channels to the left and right channels of the recorder. That's all there is to it.

To record your music, follow these steps:

1. Load Deluxe Music and the score you want to record.
2. To set the recording levels on your cassette recorder, find the spot in your musical score with the loudest music. Set this section of music so you can use it with **Repeat Play**, then play it over and over again. While the section plays, you can

adjust the volume to the maximum level your recorder can handle without distorting. Stop the score playback when you're finished.

3. When you're ready to record, make sure your cassette is rewound to the tape location where you want to start recording. If you're starting at the beginning of a cassette, make sure you advance the tape far enough to get past the leader of the cassette, so the beginning of your music isn't chopped off.
4. To start recording, press the Record key (or keys) on your cassette recorder, then use the **Play Song** command to start the score. When the score is finished, stop the recording by pressing the Stop key on the cassette recorder.

SPECIAL EFFECTS USING THE TASCAM MINISTUDIO PORTA ONE

When you use a recorder connected directly to the Amiga, you record just what the Amiga puts out through its audio ports: voices A and D coming from the left channel and voices B and C coming from the right channel with no imaging. If you want to keep the voices from sounding so separate, and would like to be able to record more than four voices playing at once, you can use a recorder like the Tascam Ministudio Porta One recorder.

The Porta One is a portable cassette recorder that records on cassettes using four tracks instead of the two you normally get with a stereo cassette recorder. It has a built-in mixing panel you can use to record the left and right signals of the Amiga on any of the four channels, or combinations of the channels. The Porta One mixing panel lets you control the volumes of each individual recording channel, and also lets you pan the sound between channels so you can blend the two distinct left and right channels coming from the Amiga. You can also use the pan controls as you record to make one of the channels coming from the Amiga seem to move from left to right.

With four available channels on the Porta One, you can record two Amiga scores on the same tape and play them back in stereo simultaneously. For example, if you want to record an eight-voice composition using just the Amiga, (something like a Beethoven symphony arrangement), you can enter four of the voices as one Deluxe Music score named BeethovenA and the other four voices as a second score named BeethovenB. You can then record

BeethovenA in stereo on tracks 1 and 2. When you're finished, you can rewind the tape and start recording BeethovenB alongside BeethovenA on tracks 3 and 4. When you play back the recording, you can hear both scores playing together simultaneously in the full eight-voice composition.

The Porta One also offers overdubbing: This means that you can mix the contents of any tape track already recorded with the contents of another recorded track and record the mixture to a third track, or you can mix one track with a new signal coming in, and record the combination to a second track. In theory, this means that you can keep overdubbing to add an infinite number of voices to the recording. For example, you can use overdubbing with your Amiga to create a recording of all of the 50 or so voices you'd need to recreate Stravinsky's "Rite of Spring" accurately. In practice, each overdub deteriorates the quality of the recording a little, so too many levels of overdubs can make the recording sound hissy and indistinct. The Porta One has an excellent noise-reduction system that lets you perform several levels of dubs, though, enough to record over 40 Amiga voices without too much distortion.

SYNCHRONIZING AMIGA SCORES

If you use a mixer or overdubbing to combine Deluxe Music scores, or if you and another Amiga owner get your computers together to perform, it's important to make sure the scores start playing at exactly the same time, or you'll go crazy listening to notes that don't start at quite the same time throughout the whole score. The best way to accomplish this is to create a "lead-in" for one of the scores—a set of beats before the score starts playing so that you can get the tempo and know exactly where to start the second score.

To create a lead-in for a score, insert one or two measures of an instrument playing every beat at the beginning of the score. For example, in a 4/4 score, you might add two measures of quarter notes at the beginning of the score as a lead-in, with a percussive instrument like a snare drum beating each note. Then, when you play the score back, you'll hear eight snare-drum beats lead into

the beginning of the music in the score. You can synchronize a second score to this by starting the second score at the exact spot where you think the ninth snare drum beat would fall: The two scores should start simultaneously. This takes some practice and attentive listening to get it just right, but if you're recording the results, the tape recorder won't mind if you stop and start over again when you miss the beat.

OTHER AVAILABLE MUSIC SOFTWARE

There are a variety of other music programs for the Amiga besides Deluxe Music that provide entirely different features and abilities. Some of these programs might have just the features you need to make the kind of music you want.

INSTANT MUSIC

Instant Music is a simple but enjoyable music program from Electronic Arts. It's perfect for music dabblers. You can enter scores up to 64 measures long on Instant Music using sampled sound instruments included on the disk, and can play them back at different speeds. Instead of using traditional notation, the scores are laid out as a graph of pitch and time, which can be a real advantage if you don't read music as you can simply draw in voices on the graph using the mouse and the pointer.

Instant Music also lets you perform music with a feature called "mousejam." Using mousejam, instant Music plays higher and lower pitches as you move the mouse pointer up and down. You can play a score and use mousejam as accompaniment. You can also limit the notes and rhythms you play with mousejam so that no matter where you move the pointer, the notes you play will sound good with the accompanying score.

SOUNDSCAPE

SoundScape is a music program from Mimetics Corporation that is actually a lot of programs in one package. It can perform a range of music tasks that make it useful to many different kinds of users, including neophyte musicians who want to play their Amiga as a

musical instrument, advanced musicians who want to tie banks of synthesizers together to create sophisticated music, and advanced programmers who want an easy way to add music to their programs.

The heart of SoundScape is a music operating system that specializes in handling music data and running music programs. When you first load SoundScape, the music operating system integrates itself with the Amiga operating system so they work together simultaneously. The music operating system contains many routines to create and play scores, handle waveform tables, and create music much like the Amiga operating system's devices and libraries take care of graphics, text, math operations, and other functions. These routines make full use of the sound software in the Amiga's operating system.

Advanced programmers can use SoundScape's music operating system as an extended version of the Amiga's operating system to create music without doing all the programming themselves. Since SoundScape's music routines run simultaneously with other Amiga programs, a programmer creating an education application, for example, can feed SoundScape some simple information about the music that he wants in his program, and SoundScape will play the music in the background while the education program runs.

For non-programmers, SoundScape comes with its own programs that use the music operating system. When you first load SoundScape, the music operating system is loaded, then several windows open on the monitor screen. Each window contains its own program. These programs all run simultaneously, and they work together so you can create some impressive music with them.

One of the simplest SoundScape programs lets you play the Amiga's keyboard like a musical instrument. Another simple program is a MIDI clock, a fancy sort of metronome that ticks out time in different tempos and subdivisions so that MIDI instruments and programs connected to it are synchronized by its signals.

Another very useful and enjoyable SoundScape program is the sampled sound editor. It lets you alter sampled sound instruments from the Mimetics SoundScape sound sampler (discussed later in this chapter) or from other programs like Instant Music or Deluxe Music. You can use the editor to select which portion of the sound to use as the sustain loop, set the amplitude envelope of the instrument, and transpose the sound to new octaves, among other things. When you're finished, the editor saves the new or revised sampled sound instruments to disk so you can use them with the other SoundScape programs or other music application programs.

The most extensive of SoundScape's programs is the MIDI recorder. It records notes and other performance events played on synthesizers attached to the Amiga through a MIDI adaptor or on the Amiga's own keyboard. The recorder has an unlimited number of tracks in its memory. You can record music on each track, and later mix the notes on individual tracks together on one track, or play back any combination of tracks through the MIDI adaptor or the Amiga's own audio channels. The MIDI recorder can receive or transmit on 16 different MIDI channels, so you can use it to record and control up to 16 MIDI devices at a time. The MIDI recorder also includes an editor that lets you look at the information recorded on any track, change it to correct any mistakes you made while playing the music, or add new notes.

Another SoundScape program, the mixing panel, controls all the other programs that run with it. It lets you connect any of the other SoundScape programs and MIDI channels with any other SoundScape program or MIDI channel. This gives you a wide range of possibilities. For example, you can use the Amiga's keyboard to record your performance on the MIDI recorder. Or you can connect the MIDI In signals to the Amiga itself so you can play the sampled sound instruments stored in the Amiga using the keyboard of an attached synthesizer. The possibilities offered by the mixing panel are numerous enough to keep you occupied for months at least, and when you consider that the modular nature of SoundScape makes it easy to add other programs to it, the possibilities become limitless.

THE IFF MUSIC STANDARD

You'll recall from Chapter 3 that Electronic Arts and Commodore-Amiga created the IFF standard to make it easy to transfer data from one program to another program. The IFF graphics standard lets you transfer pictures between programs like Deluxe Paint and Aegis Images. There are also IFF standards that apply to music: one for musical scores, and another for sampled sound instruments.

If a program uses the IFF standard to save scores and instruments, then you can exchange scores and instruments between programs that conform to the IFF standard. As an example, SoundScape and the SoundScape sound sampler both save instruments using the IFF standard, so any instruments you create with these programs can also be used with Deluxe Music and Instant Music.

ADDITIONAL SOUND HARDWARE FOR THE AMIGA

Just as you can add hardware to the Amiga to enhance its graphics, you can also add sound hardware to make your Amiga sound better and increase its music-making capabilities. The following sections describe some of the hardware you can add, ranging from external speakers to a network of external synthesizers.

SPEAKERS

The speaker in most computer monitors is no better than the tiny (and tinny) speaker in the average television set. If you've ever listened to the garbled sound of music coming from an average TV set, then you know why it pays to add some high-fidelity speakers to your Amiga system if you plan to use it for any kind of music at all and want to take advantage of its superior multi-voice sound capabilities.

Using a stereo system

If you own a good stereo system, you can connect your Amiga to the system's amplifier to hear the Amiga through the stereo speakers. It's no more difficult than connecting a cassette deck to the amplifier because the Amiga puts out a line-level audio signal—the same signal strength used by most cassette decks.

To connect your Amiga to a stereo system, first turn off both the Amiga and the stereo system. Use stereo patch cables to connect the Amiga's audio ports to a pair of stereo inputs on the back of your amplifier, just as you would connect the Amiga to a cassette deck. The inputs you use on your amplifier could be any inputs labeled for a tape deck, a compact disc player, a videotape player, or other auxiliary device. The only input that you should avoid is an input for a phonograph player; it expects a much weaker signal than the Amiga puts out, will distort the sound of the Amiga, and may even blow out your speakers if you turn up your amplifier's volume too high.

After the connection is made, turn on the power to both systems, load your music program, then play a score. Turn up the volume of the stereo system slowly. If the speakers blare out when you've barely cracked the volume, then you know the Amiga's signal is too strong. Repeat the connection process and try another pair of inputs.

Self-powered speakers

If you don't have a stereo system you want to use with your Amiga, you might consider buying a pair of self-powered speakers that contain their own amplifiers. Self-powered speakers don't have to be part of a larger sound system—you just hook them up directly to the Amiga's audio ports and turn them on. Most self-powered speakers are also small, and can fit on your desk on each side of the Amiga for good stereo separation.

The most common type of self-powered speakers are the small speakers designed to work with very small, portable cassette players. Unfortunately, most of these speakers won't work with the Amiga because they're designed to use the signal from the headphone jack of the cassette player, which has already been amplified by the player so that it's stronger than a line-level signal. Speakers designed to use just a headphone signal don't have enough amplification power to work well with the Amiga.

Some self-powered speakers are designed to work with synthesizers. They use a line-level input, have plenty of amplification, and usually have good sound. A good example of this kind of speaker is the Casio AS-20 speaker. It has a built-in graphic equalizer and chorus knobs that let you jazz up the sound, a headphone jack, and two separate inputs.

Shopping for speakers

When you go shopping for a pair of speakers for your Amiga, you'll want to check them out without dragging your Amiga from store to store. Ask the sales clerk to demonstrate the speakers by connecting them to a cassette deck or compact disc player. Make sure he doesn't connect the speakers to the headphone jack of the deck or player, but to the line-out jacks. Since cassette decks and compact disc players put out the same strength signal as the Amiga, the speakers that work with them should also work with your Amiga.

Portable cassette players

Another way to get good sound from your Amiga is to use a battery-powered portable cassette player with an amplifier and stereo speakers built in. Many of these players have jacks for a line-level input where you can plug in the Amiga. Portable cassette players have an added advantage: You can also use them to record your Amiga music.

SOUND SAMPLERS

Some of the best-sounding instruments in the music application software mentioned here are sampled sound instruments. They were created with a sound sampler (sometimes called an audio digitizer), a device that converts analog signals from a microphone, tape deck, or other sound source into a digital waveform table.

If you want to create your own sampled sound instruments, you can buy a sound sampler and start collecting sounds. Several are made to work with the Amiga. Some have their own built-in microphones; some take stereo sound samples. One important feature to look for is accompanying software that defines your sound samples and saves them to disk as sampled sound instruments.

One of the best (and most inexpensive) samplers available is the SoundScape sound sampler from Mimetics Corporation. It's a small stereo sampler that plugs into the second joystick port of the Amiga, which is an advantage over sound samplers that plug into the Amiga's expansion connector and block it so you can't add other peripherals without removing the sampler. It doesn't have a built-in microphone, but has a microphone jack and a set of stereo line-level jacks so that you can use your own microphone or other sound source.

The SoundScape sound sampler comes with software that lets you create a waveform table with the sampler, edit it, and then save it to disk. It uses the same sampled sound editor included with the SoundScape program described in the software section of this chapter. The software also lets you control the Amiga via a MIDI adaptor so you can play the Amiga and its sampled sounds with an attached external synthesizer.

THE MIDI ADAPTOR

As you recall from the last chapter, MIDI is a synthesizer industry standard that lets synthesizers exchange music information. To change the signals coming from the Amiga to a MIDI signal, you need to attach a MIDI adaptor to the serial port. Several companies market MIDI adaptors for the Amiga.

Any MIDI adaptor you get should plug directly into the Amiga serial port and should have all three MIDI connections on it: MIDI In, MIDI Out, and MIDI Thru. Watch for bargains: Some software companies may include a MIDI adaptor with their MIDI programs, others may have a great price on the MIDI adaptor alone.

EXTERNAL MIDI SYNTHESIZERS

Once you have a MIDI adaptor on your Amiga, you can add up to 16 different synthesizers and other musical devices that have MIDI interfaces. If you plan to connect a synthesizer to the Amiga to use with MIDI recorder software, the synthesizer should at least have MIDI In and MIDI Out jacks to send information to and from the MIDI recorder software on the Amiga. If you want to connect more than one synthesizer to the Amiga, then you should make sure that the synthesizers also have a MIDI Thru jack.

You can find a wide variety of MIDI synthesizers, ranging from professional synthesizers that cost tens of thousands of dollars to "toy" synthesizers that cost only a few hundred dollars. One of the best synthesizers available in a low price range is the Casio CZ-101 synthesizer.

The CZ-101 is a full-featured digital synthesizer. It has a set of 32 different instruments stored in its own memory, and it provides you with the controls to design your own instruments that you can store in small non-volatile RAM cartridges that plug into the back of the CZ-101.

The CZ-101 has a four-octave keyboard that plays up to eight notes at a time, and a pitch wheel that lets you bend the pitch of any note up and down as you play it. Unfortunately, the keyboard is smaller than a standard keyboard, and can be hard to play if you're used to a standard keyboard. Another synthesizer to consider is the Casio CZ-1000, which is identical in all respects to the CZ-101, but has a full-size keyboard and a higher price.

The CZ-101 has MIDI In and MIDI Out ports, so you can use it with Amiga MIDI recorder software. In addition, it has two MIDI modes that make it valuable in use with a MIDI recorder: You can use the CZ-101 as a single synthesizer playing up to eight notes simultaneously on one MIDI channel, or you can use it in a special mode to make the CZ-101 work like four separate synthesizers on four individual MIDI channels.

In the special mode, the CZ-101 lets you choose four different instruments from its memory, one for each of four MIDI channels. Each instrument is monophonic—that is, it only plays back one note at a time. It can't play back chords. To use the special mode, an Amiga MIDI recorder can send signals through the four different channels to play a four-part score using four individual instruments. (Since all 16 MIDI channels come through a single cable, you only need a single MIDI In connection.) Most synthesizers don't let you use more than one instrument at a time, so you gain quite a bit of versatility using the CZ-101.

You've had a chance now to use some tricks to enter Deluxe Music scores. You've also seen some of the additional software and hardware that can open up some new musical worlds to you as an Amiga user. In the next chapter, you'll learn how to use Amiga BASIC sound statements to make music or to synthesize speech.

**CHAPTER NINE
AMIGA BASIC
SOUND: MUSIC AND
SPEECH**



Amiga BASIC doesn't have nearly as many statements and functions that deal with sound as statements and functions that deal with graphics: There are a total of six sound statements if you count BEEP (which many people don't), and one sound function. But don't let their numbers fool you. Amiga BASIC sound statements are powerful, and capable of a wide variety of effects.

This chapter deals with two kinds of sound statements: statements that create music and sound effects, and statements (well, actually a statement and a function) that create speech. In addition to explaining how these statements work, this chapter also shows you how to use the statements in larger programs, and how to create musical scales and play back musical scores.

THE BEEP STATEMENT

The BEEP statement is an easy introduction to sound on the Amiga—it's very simple. It uses this format:

```
BEEP
```

When your program executes BEEP, it flashes the entire screen briefly, using inverse colors, and makes a short beep using the monitor speaker. The signal for the beep comes from the left audio port in the back of the Amiga console. It's pitched at A880, two A's above middle C on the piano. The beep uses whatever waveform is in audio channel 0, so you can change its timbre with the WAVE statement. (You can find out more about audio channels in the description of SOUND, and more about changing timbres in the description of WAVE later in this chapter.)

Since you can't change the length or pitch of BEEP's beep, BEEP isn't often used for music or sound effects. It is more commonly used as a prompt at the end of a long BASIC task to let the user know the task is finished. BEEP can call you back in from another room so you don't have to pace back and forth in front of the Amiga.

THE SOUND STATEMENT

SOUND plays a note or a rest, and controls three sound attributes: pitch, duration, and volume. It also controls which audio channel a note will play from. It uses this format:

`SOUND frequency, duration, volume, audio channel`

The frequency can be any number from 20 to 15000. It specifies the number of cycles per second (cps), which sets the pitch of the note. The duration is any number from 0 to 77; it sets the length of the note from no length at all (0) to a note that lasts 4.23 seconds (77). The volume is a number from 0 to 255. Using 0 provides no volume at all (silence), and 255 is the loudest possible volume. The audio channel is an integer from 0 to 3 that assigns the note to the audio channel of the same number.

The values for volume and audio channel are optional. If you leave them out of the SOUND statement, the note will have a default volume of 127 (halfway between silence and full volume), and it will play on audio channel 0, through the left audio port.

SETTING THE FREQUENCY

If you're familiar with pitches measured in cycles per second, then setting the pitch of a note created by the SOUND statement is very straightforward. You just enter the frequency of the pitch you want. If, like most people, you are more familiar with pitch names like A, B, C, D, E, F, and G, then a quick look at how frequency is related to pitch will help you understand how to set the pitch for the note you want.

To describe pitches in different octaves, pitch names traditionally have a subscript numbered from 0 to 10 to show the octave in which the pitch falls. Therefore, E_0 is the E in octave number 0, the lowest octave, and Bb_{10} is the B \flat in the 10th octave, the highest octave. Don't confuse this Amiga BASIC system with systems used to describe pitch names in music application programs, where the octave numbers sometimes start at a higher octave and can't describe pitches as high and as low as you are reading about in this chapter.

The range of frequencies you can use with the SOUND statement is 20 cps to 15000 cps. That is a range from E_0 (the third E below the bottom of the bass clef) to Bb_9 (the fifth B \flat above the treble clef), a range of about eight and a half octaves. Compare that to the range of a piano, which is six and a half octaves.

To get an idea of how the frequencies relate to specific pitches, consider some familiar examples. The C in the middle of the piano keyboard, usually called middle C, is C_4 and has a frequency of

261.63 cps. The A above that is A₄, the note that orchestras use to tune. Musicians know it as A440, so it's not surprising that its frequency is 440 cps.

Figure 9-1

A list of pitches and their frequencies that can be used with the SOUND statement.

Pitch	Frequency	Pitch	Frequency	Pitch	Frequency
E ₀	20.602	G ₃	196.00	A# ₆	1864.7
F ₀	21.827	G# ₃	207.65	B ₆	1975.5
F# ₀	23.125	A ₃	220.00	C ₇	2093.0
G ₀	24.500	A# ₃	233.08	C# ₇	2217.5
G# ₀	25.957	B ₃	246.94	D ₇	2349.3
A ₀	27.500	C ₄	261.63	D# ₇	2489.0
A# ₀	29.13	C# ₄	277.185	E ₇	2637.0
B ₀	30.868	D ₄	293.66	F ₇	2793.8
C ₁	32.703	D# ₄	311.13	F# ₇	2960.0
C# ₁	34.648	E ₄	329.63	G ₇	3136.0
D ₁	36.708	F ₄	349.23	G# ₇	3322.4
D# ₁	38.891	F# ₄	369.99	A ₇	3520.0
E ₁	41.203	G ₄	392.00	A# ₇	3729.3
F ₁	43.654	G# ₄	415.30	B ₇	3951.1
F# ₁	46.249	A ₄	440.00	C ₈	4186.0
G ₁	48.999	A# ₄	466.16	C# ₈	4434.9
G# ₁	51.913	B ₄	493.88	D ₈	4698.6
A ₁	55.000	C ₅	523.25	D# ₈	4978.0
A# ₁	58.270	C# ₅	554.37	E ₈	5274.0
B ₁	61.735	D ₅	587.33	F ₈	5587.7
C ₂	65.406	D# ₅	622.25	F# ₈	5919.9
C# ₂	69.296	E ₅	659.26	G ₈	6271.9
D ₂	73.416	F ₅	698.46	G# ₈	6644.9
D# ₂	77.782	F# ₅	739.99	A ₈	7040.0
E ₂	82.407	G ₅	783.99	A# ₈	7458.6
F ₂	87.307	G# ₅	830.61	B ₈	7902.1
F# ₂	92.499	A ₅	880.00	C ₉	8372.0
G ₂	97.999	A# ₅	932.33	C# ₉	8869.8
G# ₂	103.83	B ₅	987.77	D ₉	9397.3
A ₂	110.00	C ₆	1046.5	D# ₉	9956.1
A# ₂	116.54	C# ₆	1108.7	E ₉	10548.1
B ₂	123.47	D ₆	1174.7	F ₉	11175.3
C ₃	130.81	D# ₆	1244.5	F# ₉	11839.8
C# ₃	138.59	E ₆	1318.5	G ₉	12543.9
D ₃	146.83	F ₆	1396.9	G# ₉	13289.8
D# ₃	155.56	F# ₆	1480.0	A ₉	14080.0
E ₃	164.81	G ₆	1568.0	A# ₉	14917.2
F ₃	174.61	G# ₆	1661.2		
F# ₃	185.00	A ₆	1760.0		

If you double the frequency of a pitch, you get a pitch one octave higher. If you halve the frequency, you get a pitch one octave lower. For example, if you double A_4 's frequency of 440 cps, you get 880 cps, which is A_5 , an A one octave higher. 220 cps, half of 440 cps, is A_3 , an A one octave lower than A_4 .

Figure 9-1 gives you a list of the pitches SOUND can create, followed by their frequencies. This list of pitches is a tempered scale, a standard used by most musicians, and is tuned so that A_4 is at 440 cps.

To be able to hear all these pitches, you need to connect your Amiga to a good audio system. If you use only the Amiga's monitor speaker to produce sounds, you probably won't be able to hear any pitches under 100 cps or over 8000 cps unless you change the waveform of the pitch with the WAVE statement (discussed later in this chapter).

SETTING THE DURATION

The duration value of the SOUND statement can be any number from 0 to 77. The larger the number, the longer the note lasts. A value of 0 means the note has no length. A value of 1 makes the note last a little more than 1/20 of a second. A value of 18.2 makes the note last exactly one second, and the maximum value of 77 makes the note last about 4.23 seconds. For example, this statement plays an A_{440} for one second:

```
SOUND 440, 18.2
```

If you figure out the length in seconds of the note you want to sound, you can multiply it by 18.2 to come up with the duration value to use in the SOUND statement. For example, a 1/2-second sound would use 9.1 as its duration value. To make a sound last longer than 4.23 seconds, you can use two or more SOUND statements with the same pitch, volume, and audio channel values, one after the other. They will run together to create one long note. For example, these two statements play one F_4 that lasts for seven seconds:

```
SOUND 349.23, 72.8
```

```
SOUND 349.23, 54.6
```

The first SOUND statement plays a 4-second note, the second SOUND statement plays a 3-second note. (Since no volume or audio channel is specified in the statements, the notes both have a default volume of 127 and play on audio channel 0.)

SETTING THE VOLUME

Choosing a SOUND volume is a simple matter of choosing a number from 0 to 255. If you choose 0, you won't be able to hear the note you set, so it doesn't matter what frequency you specified. The silence will last as long as the duration you set, so you can use 0 volume to create a rest. For example, the middle SOUND statement below puts a 1/2-second rest in between two surrounding notes that play at full volume:

```
SOUND 440, 18.2, 255  
SOUND 440, 9.1, 0  
SOUND 392, 18.2, 255
```

You can use volume values between 0 and 255 to play a note at different volumes. Just remember that the volume you set is affected by the volume control on your monitor or stereo system. If you turn the volume control down to half of its previous volume, all the notes created with the SOUND statement will be halved in volume.

CHOOSING AN AUDIO CHANNEL

You can choose any one of the Amiga's four internal audio channels to play a note created with a SOUND statement. By using four different SOUND statements, one for each channel, you can sound four notes simultaneously. For example, these four SOUND statements play a C major chord:

```
SOUND 261.63, 20, 255, 0  
SOUND 329.63, 20, 255, 1  
SOUND 392, 20, 255, 2  
SOUND 523.25, 20, 255, 3
```

The outputs of the four internal audio channels are mixed together before they are passed through the two audio ports on the back of the console to an outside speaker. Channels 0 and 3 come out of the left audio port, and channels 1 and 2 come out of the right audio port. If you have stereo speakers attached to your Amiga, you can make a note sound in either the left or right speaker by choosing the appropriate audio channel in the SOUND statement.

Each internal audio channel has its own waveform that determines the timbre of the notes created with the SOUND statement

on that channel. You can change the waveform with the `WAVE` statement, discussed later in this chapter. If you set different waveforms for each of the audio channels, then you can choose the timbre of the note a `SOUND` statement plays by specifying the audio channel with the waveform you want.

SYNCHRONIZING SOUNDS

As you've seen in the previous example, it's possible to produce notes from the four internal audio channels simultaneously by using four `SOUND` statements. It can be difficult to make all four notes start at exactly the same time, however, if you have entered any other `BASIC` statements in between the `SOUND` statements. Since Amiga `BASIC` steps through the statements one at a time, there can be a delay between `SOUND` statements, which can throw the individual notes out of sync. Amiga `BASIC` has two statements that help you synchronize notes: `SOUND WAIT` and `SOUND RESUME`.

THE SOUND WAIT AND SOUND RESUME STATEMENTS

`SOUND WAIT` is used along with `SOUND RESUME` to synchronize the Amiga's audio channels so they start playing notes at exactly the same time. This ensures that notes in multiple-voice music will play together without the risk of one voice getting off from the beat. `SOUND WAIT` prevents sounds generated by the `SOUND` statement from going to the speaker—instead, the sounds are held in a "queue," which can be thought of as a holding tank in the Amiga's memory. Each audio channel has its own queue that it uses to store up to twelve notes and rests. A single `SOUND WAIT` statement starts all four queues working. If you attempt to store more than twelve notes and rests in a channel's queue (twelve `SOUND` statements sent to one channel), you will get an out-of-memory error. `SOUND WAIT` uses this simple format:

```
SOUND WAIT
```

When Amiga `BASIC` encounters this statement, any sounds created by subsequent `SOUND` statements are held in the queues until you release them with a `SOUND RESUME` statement. `SOUND RESUME` sends the contents of the queues to the speakers at exactly the same time, simultaneously sounding all the voices

through the speakers. The queues are cleared as the stored notes are played. The format of SOUND RESUME is simply:

SOUND RESUME

To see how SOUND WAIT and SOUND RESUME work, try the following program. It creates five notes for each of three audio channels: channel 0, channel 1, and channel 2. There is a loop just after the SOUND statements for channel 1 that extends the length of time the program takes to get to the channel 2 SOUND statements. This makes it easy to hear a time lag between voices. There is also an infinite loop at the end of the program to keep the List window from appearing. If a window appears while notes are playing, it can delay one of the audio channels and ruin the synchronization of the four voices.

First enter and run this program without typing in the SOUND WAIT and SOUND RESUME statements shown:

```
SOUND WAIT
SOUND 261.63, 12, 255, 0
SOUND 196, 12, 255, 0
SOUND 261.63, 12, 255, 0
SOUND 196, 12, 255, 0
SOUND 261.63, 48, 255, 0
SOUND 261.63, 12, 255, 1
SOUND 293.66, 12, 255, 1
SOUND 329.63, 12, 255, 1
SOUND 293.66, 12, 255, 1
SOUND 261.63, 48, 255, 1
FOR t = 1 TO 1000: NEXT t
SOUND 329.63, 12, 255, 2
SOUND 349.23, 12, 255, 2
SOUND 392, 12, 255, 2
SOUND 349.23, 12, 255, 2
SOUND 329.63, 48, 255, 2
SOUND RESUME
Loop: GOTO Loop
```

You should be able to hear one voice (channel 2) starting and ending much later than the other two voices. Now insert the SOUND WAIT and SOUND RESUME statements in the program and run it again. All three voices should start and end at the same time, since SOUND WAIT queues up the notes, and then SOUND RESUME starts them playing at the same time.

THE WAVE STATEMENT

You can use the WAVE statement to create a new waveform table for each of the four audio channels. Changing a channel's waveform table changes the timbre of any sound played through that channel. WAVE uses this format:

```
WAVE audio channel, integer array name
```

The audio channel is an integer from 0 to 3 that specifies the audio channel to which you want to assign a new waveform table. The integer array name is the name of the integer array that contains the data for the waveform table. You have to create the array before you use the WAVE statement.

You can substitute the specification *SIN* in place of the integer array name to make WAVE automatically create a sine waveform table for the audio channel you specify. All the audio channels use a sine waveform as a default waveform if you don't change their waveform tables with a WAVE statement. Using *SIN* in the WAVE statement returns the audio channel to its default waveform.

CREATING A WAVEFORM TABLE

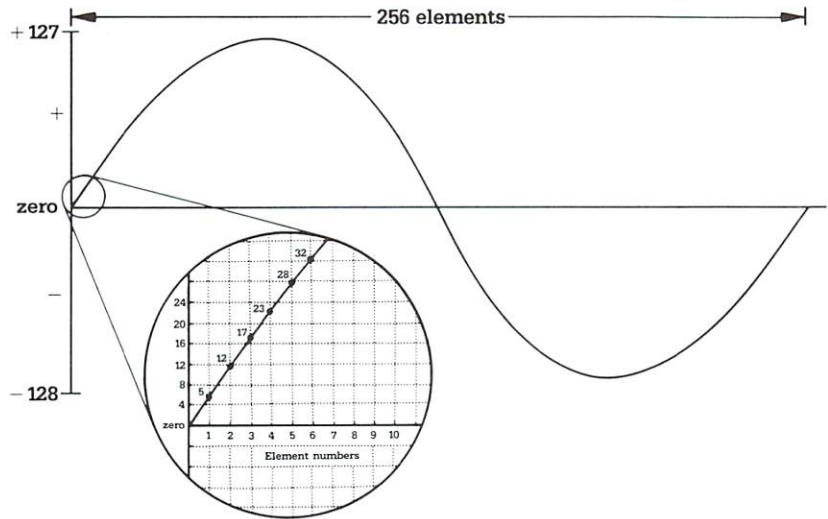
To create a waveform table for the WAVE statement, your first step is to dimension an array. WAVE uses an integer array; any other type of array will generate a *Type Mismatch* error in your program. The array must have at least 256 elements in it, but it can be longer if you want to define a more detailed waveform within a waveform table. As an example, the following statement dimensions an integer array named *form%* to include 256 elements (0 to 255). The % signifies that it is a short (2-byte) integer array:

```
DIM form%(255)
```

The next step is to fill the elements of the array with waveform table data. Each of the 256 elements in the array must be an integer between -128 to 127, inclusive. The array elements in order from lowest to highest describe the shape of a waveform from left to right in a waveform graph. To get an idea of how this works, look at Figure 9-2 on the next page. It depicts a waveform on a graph, and then shows the first section of the waveform magnified, with the first 6 waveform table values plotted. These first 6 values go into the first 6 elements of the integer array, followed by the next 250 elements needed to describe the full waveform.

Figure 9-2.

The graph of a full sound waveform, showing an enlargement of the beginning of the waveform and the first six waveform table values.



To fill the elements of the array with waveform table values, you can set up a FOR...NEXT loop to read a value for each element with READ and DATA statements, or you can create a FOR...NEXT loop with a mathematical formula that creates the waveform values as the loop progresses.

Reading waveform data into the waveform array

To read waveform data in the waveform array with READ and DATA statements, you first have to have at least 256 waveform table values. One way to get them is to first draw the waveform on graph paper, then measure the amplitude of the waveform at 256 different points along the horizontal axis of the graph, as you saw above in Figure 9-2. Be sure to scale the vertical axis of the graph so that the waveform goes no higher than 127 and no lower than -128. Enter the values in DATA statements in the order you read them from left to right.

The following statements show a FOR...NEXT loop that runs 256 times and reads a waveform value into the *form%* array each time. The DATA statements following the loop contain the waveform data, but not all the DATA statements are shown here (you don't really want to read 256 values, do you?):

```
DIM form%(255)
FOR i = 0 TO 255
  READ form%(i)
NEXT i
DATA -112, -112, -111, -110, -108, -105, -103, -94, -83 etc.
```

Calculating simple waveforms

A much shorter way to create a waveform table is to calculate the waveform with a simple formula inside a FOR...NEXT loop. This works very well for simple waveforms such as square waves, sawtooth waves, and notch waves. (You can see examples of all these waves in Figure 9-4 on the next page.) These statements show how to create a sawtooth waveform:

```
DIM form%(255)
FOR i = 0 TO 255
  form%(i) = 127 - i
NEXT i
```

This short loop would create the waveform you see in Figure 9-3 if *form%* is applied using a WAVE statement.

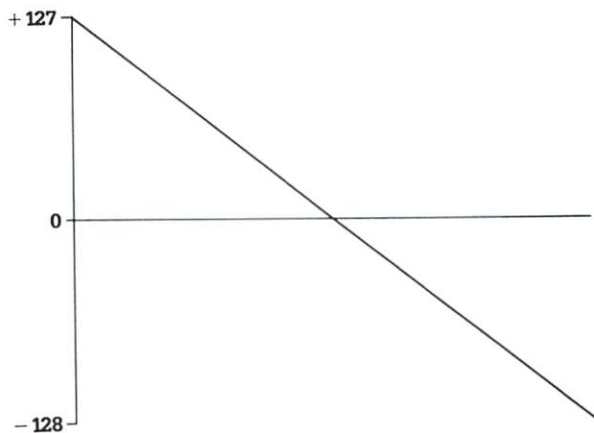


Figure 9-3.

A sawtooth waveform can be created with a simple formula.

Here are some other simple formulas and program steps that will create more simple waveforms with distinct timbres. Just put them in the same loop shown above, replacing the sawtooth formula with the formula you want. You can see the waves created by the formulas in Figure 9-4, except for the random waveform, which turns out a different waveform each time you use it.

A square waveform:

```
IF i < 128 THEN form%(i) = 127 ELSE form%(i) = -128
```

A triangle waveform:

```
IF i < 128 THEN form%(i) = (2*i) - 128 ELSE form%(i) = 383 - (2*i)
```

A notch waveform:

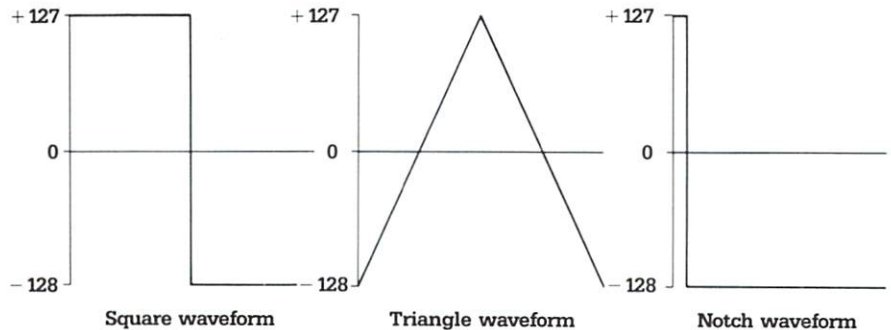
```
IF i < 5 THEN form%(i) = 127 ELSE form%(i) = -128
```

A random waveform:

```
form%(i) = 127 - INT(256 * RND)
```

Figure 9-4.

Waveforms created using simple formulas.



You can no doubt come up with other formulas that create waveforms with distinct timbres. Any formula is safe to use as long as its results are always integers that are not lower than -128 or higher than 127 . It's also important to make sure that at least some of the values come close to the limits of the -128 to 127 integer range. If they don't, the volume of the sound won't be very high. For example, if you create a waveform that goes no higher than 6 and no lower than -3 , you're going to have trouble hearing it at all, regardless of the volume setting of your monitor speaker.

ASSIGNING A WAVEFORM TABLE TO AN AUDIO CHANNEL

Once you've created a waveform table and assigned it to a waveform array, it's a simple matter to assign the table to an audio channel. Just specify the audio channel and the name of the array

in a WAVE statement. To hear the waveform, you can use a SOUND statement to play a note using the same audio channel. The following short program creates a notch waveform in channel 0, then plays an A₄ for one second at full volume in channel 0 so you can hear the waveform:

```
DIM form%(255)
FOR i = 0 TO 255
  IF i < 5 THEN form%(i) = 127 ELSE form%(i) = -128
NEXT i
WAVE 0, form%
SOUND 440, 18.2, 255, 0
```

If you want to hear some other waveforms, try replacing the notch waveform formula in the program with some of the other waveform formulas.

CREATING A MUSICAL SCALE ARRAY

If you use individual SOUND statements to create a series of notes, you'll find it can get tedious to look up a pitch and use the correct frequency for each note. You can simplify your task a great deal by creating an array that stores the frequency for each pitch of the tempered scale; then all you have to do is specify the array name and element number of each note you want to use in a SOUND statement. The following program creates an array for each pitch of the tempered scale, and then plays all the pitches in the array from bottom to top:

```
DIM pitch(118)
halfstep# = 2^(1#/12#)
frequency# = 13.75
FOR i = 1 TO 8
  frequency# = halfstep# * frequency#
NEXT i
FOR i = 4 TO 116
  frequency# = halfstep# * frequency#
  pitch(i) = frequency#
NEXT i
FOR i = 4 TO 116
  SOUND pitch(i), 4, 255
NEXT i
```


Chances are you won't hear all the pitches when you run the program, since some of them are beyond the range of the monitor speaker, and the default sine waveform that plays the notes is hard to hear in the low range. If you assign a notch waveform to channel 0 with a WAVE statement at the beginning of this program, you should be able to hear the low notes much better.

Crucial to this program is the fact that if you multiply a frequency by the 12th root of 2 (which is $2^{1/12}$ in Amiga BASIC), the resulting frequency is one half step higher than the original frequency. The variable *halfstep#* is set to the 12th root of 2 in the second line of the program, and is used in lines 5 and 8 as a multiplier to create half step increases in frequency. The program starts with a base frequency of 13.75, one octave below A_0 , to tune the entire scale at 440 cps for A_4 (13.75 is 440 divided by 2 five times, and is therefore five octaves below A_{440} —recall that halving a frequency lowers it by one octave). The variable *frequency#* holds the current frequency values. These two variables are both double-precision variables (specified by the "#") and use double-precision constants in their calculations to ensure accuracy of intonation over many different multiplications.

The first loop in the program multiplies *frequency#* by *halfstep#* to create eight half step pitches, but doesn't store them in the pitch array, since they're all below 20 cps, and cannot be used in the SOUND statement. The next loop, which steps from 4 to 116, continues multiplying *frequency#* by *halfstep#* to create new half-step frequencies, starting where the last loop left off. It stores them in the successive elements of the array *pitch*, starting at element 4, the lowest pitch, and ending with element 116, the highest pitch. The series starts at 4 because the lowest pitch in the array is E_0 , which is the lowest pitch possible in Amiga BASIC. By numbering this pitch 4, all the following C's, which begin each octave of pitches, are numbered as multiples of 12. This makes it easier to remember the elements of the pitch array, as you'll see later.

The last loop at the end of the program plays the pitches stored in the array, stepping through pitches from *pitch(4)* to *pitch(116)*.

To determine which pitch in an array to use in a SOUND statement, multiply the octave number of the pitch you want by 12, then add a number from this chart for each different pitch:

Figure 9-5.

Traditional music notation showing the positions of notes and their names.

C	0	F# or Gb	6
C# or Db	1	G	7
D	2	G# or Ab	8
D# or Eb	3	A	9
E	4	A# or Bb	10
F	5	B	11

For example, A_4 would be pitch element number 57 (4 times 12, plus 9). To play an A_4 for one second, you would use:

```
SOUND pitch(57), 18.2
```

The real utility of a pitch array becomes apparent when you want to play a piece of music using SOUND statements.

PLAYING A MUSICAL SCORE

Any piece of music you want to create on the Amiga is likely to contain quite a few notes. If you create the notes with one SOUND statement for each note, you're going to have a very long program and a lot of typing to do. An easier solution is to use a SOUND statement within a loop that reads values for pitch, duration, volume, and voice number from DATA statements that contain data for the score. The following program lines do just that. They play a short four-voice flourish of music that starts softly and gets louder at the end.

The following program lines are not a complete program; they need a pitch array like the one in the previous example to work correctly. To hear the flourish, substitute the following program lines for the last three lines of the pitch-array example shown on page 233, then run the new program.

```
SOUND WAIT
FOR i = 0 TO 3
    SOUND 30, 5, 0, i
NEXT i
SOUND RESUME

Loop:
    FOR i = 0 TO 3
        READ pitch(i), octave(i), duration(i), volume(i)
        IF pitch(i) = -1 GOTO KillTime
        SOUND pitch((12*octave(i))+pitch(i)), 32/duration(i), volume(i), i
    NEXT i
    GOTO Loop
```

(continued)

```

KillTime:
    FOR t = 1 TO 4000: NEXT t: END

MusicData:
    DATA 7,4,16,63, 7,3,4,63, 4,3,4,63, 0,3,4,63
    DATA 0,5,16,87, 7,3,4,159, 2,3,4,159, 11,2,4,159
    DATA 11,4,16,111, 7,3,4,255, 4,3,4,255, 0,3,4,255
    DATA 9,4,16,135, 0,0,16,0, 0,0,16,0, 0,0,16,0
    DATA 9,4,16,159, 0,0,16,0, 0,0,16,0, 0,0,16,0
    DATA 9,4,16,183, 0,0,16,0, 0,0,16,0, 0,0,16,0
    DATA 9,4,16,207, 0,0,16,0, 0,0,16,0, 0,0,16,0
    DATA 9,4,16,231, 0,0,16,0, 0,0,16,0, 0,0,16,0
    DATA 9,4,16,255, 0,0,16,0, 0,0,16,0, 0,0,16,0
    DATA -1,-1,-1,-1

```

The first loop in these program lines, sandwiched between a SOUND WAIT and a SOUND RESUME statement, causes each of the four voices to play a rest of exactly the same duration. The SOUND WAIT and SOUND RESUME statements synchronize the four rests so they all begin at exactly the same time. While these rests are playing, the program reads data into the *pitch* array and uses them to create notes with the SOUND statement that follows. Because the rests are still playing, these notes are held in readiness for each audio channel until the rests end. Since the rests end at exactly the same time, the notes from the later SOUND statements all start at the same time, so the four voices of the flourish are synchronized.

This method of using a short rest loop at the beginning of a score to synchronize the notes, instead of using multiple SOUND WAIT and SOUND RESUME statements, is preferable for long musical scores. This is because the note queue created for each voice by SOUND WAIT only holds 12 notes at a time per voice, so you would have to use many contiguous SOUND WAIT and SOUND RESUME statements to synchronize a long score. If you run a program containing a contiguous series of SOUND WAIT and SOUND RESUME statements, the music will “hiccup” when one queue ends and another begins. The rest-loop method used in this program isn’t really needed for a flourish this short, but it will come in very handy if you want to replace the DATA statements shown here with your own DATA statements for a longer piece of music.

After the synchronization loop is an infinite loop named *Loop*:. In it is a FOR...NEXT loop that makes one pass for each audio channel (from 0 to 3). Each time the FOR...NEXT loop makes a pass, it uses a READ statement to read four specifications—pitch, octave, duration, and volume—and stores them in four variables. In the next line, it checks to see if the pitch is -1 , a value that serves as a flag at the end of the music data to show that the score is ended. If it is -1 , the program exits the loop and ends.

If it's not the end of the score, the next line of the FOR...NEXT loop multiplies the octave value by 12 and adds the scale pitch value to come up with the correct element of the pitch array, which it uses as the frequency in the SOUND statement. The note's duration is set by dividing a set value, arbitrarily set here at 32, by the duration value that is read from the DATA statement. The note's volume is set by using the volume value that is read directly from the DATA statement.

After four passes (once for each audio channel), the FOR...NEXT loop ends, and the *Loop*: loop repeats, running the FOR...NEXT loop again and again until it encounters the -1's read from the DATA statements. When it does, the program goes to *KillTime*:, and a short FOR...NEXT loop there kills a little time before the program ends so the List window doesn't appear too soon. If the List window appears while the music is playing, it makes the notes falter.

The note data in the DATA statements are organized so you have four notes in each DATA line, one for each voice, in voice order 0, 1, 2, and 3. Each note has four values—pitch, octave, duration, and volume. The pitch is an integer from 0 to 11 that corresponds to a pitch name; A, F, E \flat , etc., as discussed earlier in the section on pitch arrays. The octave is an integer from 0 to 9 corresponding to the octave subscript that usually follows a pitch name.

The duration is a value corresponding to the length of note you want played: 1 for a whole note, 2 for a half note, 4 for a quarter note, 8 for an eighth note, 16 for a sixteenth note, etc. These values don't set a specific length of time by themselves: They are used by the SOUND statement to divide a number (in this example 32) that sets the duration of a whole note. By changing this number, you actually change the speed of the music, because all the duration values in the DATA statements are divided into this number to determine each note's length. For example, change the 32 in the SOUND statement to 64 and run the program again, and you'll hear the flourish played at half the previous speed. If you change it to 16, you'll hear it played twice as fast.

The last value for each note is volume, which sets the volume in the SOUND statement.

Each DATA statement contains four notes, one for each voice. Within each DATA statement, each note's data is separated from the next note's data by a space, so you can easily distinguish the four notes on each line. Voice 0 is the group immediately following the word DATA on each line. Note that the duration value for voice 0's data on each line is 16, which indicates sixteenth notes. Voices 1, 2, and 3 all play quarter notes (as indicated by the 4 for their duration values), and only require the first three DATA statements

to hold their note data. Because voice 0 has more individual notes than the other three voices, the values in the remaining DATA statements for the other voices become 0, 0, 16, 0, effectively making a rest that's a sixteenth note in length to hold a place for those voices so voice 0 can continue to read its data. (If you specify a duration of 0, the program will try to use the value to divide with, and you'll get a division-by-zero error.)

At the end of the DATA are four -1's to signify the end of the score. You have to have four -1's because the program reads data in chunks of four.

If you use this system of READ and DATA statements to store notes, you can create new and longer scores just by using different data in the DATA statements. If you precede the program with WAVE statements to change the timbre of the different audio channels, you can give each voice its own unique timbre to play the notes. Experiment, and have fun!

GENERATING SPEECH

The Amiga's system software includes libraries and devices that generate speech using the same hardware and much of the same software routines that the Amiga uses to create music and sound effects: It speaks words using waveforms created from waveform tables, and plays the waveforms through the same four internal audio channels. You don't have to define waveforms and use exotic SOUND statements, because Amiga BASIC provides two powerful tools, the TRANSLATE\$() function and the SAY statement, that you can use to take direct advantage of the Amiga's built-in speech-generation capabilities. These provide you with a simple yet powerful means of adding speech to a program.

SAY is the Amiga BASIC statement that instructs the Amiga to speak and tells it what to say. SAY requires all the words you want it to speak to be spelled out phonetically, using a special system of phoneme codes described in Appendix H of the Amiga BASIC manual. If you don't want to spell out everything phonetically, you can use the TRANSLATE\$() function, which takes a string of English text and translates it into the phoneme codes needed by the SAY statement.

THE TRANSLATE\$() FUNCTION

The TRANSLATE\$() function translates English words into a string of phoneme codes for use by the SAY statement. The TRANSLATE\$() function uses this format:

```
TRANSLATE$( "text string" )
```

The text string can be any string of characters that's legal to use in Amiga BASIC, or a string variable representing the text to be spoken. A text string must be specified using quotation marks within parentheses; a string variable requires only parentheses.

Since TRANSLATE\$() is a function, it doesn't do anything by itself. The string of phoneme codes it returns should be assigned to a string variable, used directly with the SAY statement or printed out using a PRINT statement. Using TRANSLATE\$() directly with SAY is very convenient. For example,

```
SAY TRANSLATE$( "I speak English very well, thank you." )
```

speaks the sentence shown in quotes.

To see the phoneme code that TRANSLATE\$() returns, you can use the function with PRINT. For example,

```
PRINT TRANSLATE$( "I speak English very well, thank you." )
```

prints out the phoneme code that SAY used in the last example:

```
AY SPIY4K IY3NXGLIHS VEH1RIY WEH4L, THAE4NXK YUW.
```

The string is returned in capital letters because the SAY statement requires phonetic spelling in capital letters. TRANSLATE\$() returns a capitalized string of phonetic spelling that SAY can use, so don't worry about capitalization if you use TRANSLATE\$() within a SAY statement.

If you print out an English sentence as phonemes, you can go in and alter it slightly to your satisfaction, and then use it directly with SAY to fine-tune pronunciation and inflection. (See Appendix H of the Amiga BASIC manual.) For example, you might want to de-emphasize the word "thank" near the end of the sentence. To do so, use a PRINT statement with the TRANSLATE\$() function to print out the phoneme string, then type the string into a SAY statement, deleting the 4 in *THAE4NXK*. This will accent the word by removing its inflection value. A word of caution: If the changes you make to translated strings of text do not conform to the guidelines specified in Appendix H of the Amiga BASIC manual, you might get an error message, or the string may not be spoken.

USING PUNCTUATION

When you use English strings with TRANSLATE\$(), you can use punctuation marks to alter the inflection of the spoken text. They affect speech inflection as follows:

- **Period:** You should use a period at the end of every sentence. It makes the voice pitch drop to indicate the end of the sentence. For example, try:

```
SAY TRANSLATE$("This is the end.")
```

- **Question mark:** A question mark at the end of a sentence works just like a period—it makes the pitch drop. For example,

```
SAY TRANSLATE$("This is the end?")
```

sounds the same as the last statement.

- **Comma:** A comma between words raises the final pitch of the word it follows and puts a pause between the two words. A comma at the end of a sentence will make it rise in pitch like a question, and should be used in place of a question mark for a question. For example, try:

```
SAY TRANSLATE$("Well, should I stop now,")
```

- **Parentheses:** Putting parentheses around a phrase will separate the phrase from the rest of the sentence with a lift in pitch and a small wait at each end of the phrase. For example, listen to:

```
SAY TRANSLATE$("I think (I really do) that I should stop now.")
```

- **Dashes:** Connecting words with dashes will cause those words to be spoken with much less inflection than words separated by spaces. For example, try:

```
SAY TRANSLATE$("I'm so tired I'm-losing-my-inflection")
```

ALTERNATE SPELLINGS FOR CORRECT PRONUNCIATION

On occasion, the English language gets the better of TRANSLATE\$() and some strange pronunciations come out of the SAY statement. This is particularly true of names and English words that use foreign pronunciations. Of course, you can always

create the words with the phoneme codes, but you might find it easier to try some creative misspelling with the TRANSLATE\$() function instead.

For example, TRANSLATE\$() has trouble with the name Michael, which it pronounces more like "Mitch ay ell." A simple misspelling will correct the pronunciation. Try both of these SAY statements to hear the difference:

```
SAY TRANSLATE$("Michael")
SAY TRANSLATE$("Mykul")
```

A little experimentation will help you understand just how TRANSLATE\$() pronounces English so you can get around its occasional limitations. All in all, though, you should be impressed with what the function can do. Any function which correctly pronounces the "o" in "women" is certainly worthy of respect.

THE SAY STATEMENT

You use the SAY statement to tell the Amiga what to say and how to say it. SAY uses this format:

```
SAY phoneme code string, specification array name
```

The phoneme code string is a string of capital letters and numbers which spells out words phonetically using the phoneme codes described in Appendix H of the Amiga BASIC manual, which is the same format the TRANSLATE\$() function uses to return a phonetic string. The specification array name is optional; it gives the name of an integer array that stores nine different values to specify how SAY will speak. If no specification array name is given, the SAY statement will speak the phoneme code string using the default male voice.

PHONEME CODES

English is a tricky language to spell and pronounce. The same letter in two different words can be pronounced two different ways, as you can hear by listening to the "c" in "cycle" and the "c" in "call." To accurately tell SAY what you want it to pronounce, you must break up the words into separate sounds, called phonemes, and use the Amiga's phoneme codes to spell out each phoneme. The easiest way to do this is to let the TRANSLATE\$() function do the work for you. However, since TRANSLATE\$() doesn't always interpret words correctly and its pronunciation rules are limited

to the English language, you may want to use phonetic spellings in place of the TRANSLATE\$() function to make SAY speak with the greatest possible accuracy.

Appendix H of the Amiga BASIC manual goes into great detail about how to spell with phoneme codes. Although you can use the TRANSLATE\$() function to create the phoneme codes for you, taking time to learn the codes yourself will be worth it if you want SAY to speak with the greatest possible accuracy. You can use the phoneme codes to specify precise accents, pauses, and emphasis you want to put in a sentence, or to make SAY speak in a foreign language. For example, the following statement says "Wie geht es Ihnen?", German for "How are you?":

```
SAY "VIY2 GEY7T IX2S IY3NIXN."
```

THE SPECIFICATION ARRAY

If you don't want to use the default voice that SAY speaks with, you can change the quality of SAY's speech by setting nine different speech aspects. Instead of including a long list of nine specifications at the end of the SAY statement, you would store the specifications in a short array that SAY reads when you give it the array name. The array must be an integer array (indicated by a % symbol at the end of the variable name), and it has to have 9 elements. The specification values are stored in elements 0 through 8, and control the pitch, inflection, speaking rate, volume, and other aspects of speech. The elements are:

Element number	Speech aspect controlled
0	Base pitch of the voice
1	Inflection choice
2	Speaking rate
3	Voice-gender choice
4	Sampling frequency
5	Volume
6	Channel-assignment choice
7	Synchronous-speech choice
8	Multiple-SAY choice

Setting the base pitch

Element number 0 of the array is an integer from 65 to 320 that gives the base frequency in cps of the voice's pitch. The default value is 110, which is an A₂. When SAY speaks, it usually adds inflection to the words, so the base pitch is only a central pitch. Inflection moves the pitch up and down around the central pitch.

Choosing inflection or monotone

Element number 1 of the array is either a 0 or a 1, and is used to specify speech with inflection, or monotone speech without inflection. The default is 0, speech with inflection. A 1 specifies monotone speech.

You'll probably want to use inflection most of the time to add life to the Amiga's utterances. If you want SAY to keep a steady base pitch, then you can use monotone. Monotone also does a great old-fashioned robot imitation.

Setting the speaking rate

Element number 2 of the array is an integer from 40 to 400 that specifies the rate in words per minute that SAY uses to speak. The default rate is 150 words per minute, a normal speaking speed. A rate of 400 is great for reciting tongue twisters; pushing the rate down below 100 will produce a torturous drawl.

Choosing the voice gender

Element number 3 of the array chooses the gender of SAY's speech. It's either a 0 or a 1. The default, 0, chooses a male voice, and 1 chooses a female voice. When you change the gender, you change the way SAY forms its phonemes, and so change the characteristic sound of its speech to sound more like the vocal chambers of either a man or a woman. The male voice is designed to work best with the default base pitch, 110 cps. If you choose a female voice, you should also change the base pitch to 220 cps or higher, an octave jump up, since most women speak about an octave higher than men.

Setting the sampling frequency

Element number 4 specifies the sampling frequency of SAY's voice in cycles per second. The frequency can be any integer in the range 5000 to 28000. The default frequency is 22000. The sampling frequency sets the rate at which the Amiga plays back the waveform table used to create a speaking voice. You might think of it as the audio equivalent of graphics resolution. In general, the higher the frequency, the smoother the voice (although a setting too high will produce a "squeaky" voice), and the lower the frequency, the rougher and more gravelly the voice.

Sampling frequency and base pitch are interlinked. As you may recall from Chapter 7, any time you speed up the rate you use to play back a waveform table, you increase the pitch of the sound coming from that table. Similarly, as you increase the vocal sampling frequency, the pitch goes up. As you decrease it, the pitch goes down. Before you change the sampling frequency to change the character of SAY's voice, you should first find the

general voice quality you want by using the default pitch of 22000 and changing the other voice attributes. Once you have the voice quality, reset the base pitch to find the new pitch you want.

Setting the volume

Element number 5 sets the volume of SAY. It can be any whole integer from 0 to 64. 0 is no volume (silence). 64 is full volume, and values between 0 and 64 set intermediate volumes. The default volume is 64.

Choosing the channel assignment

Element number 6 of the array gives you 12 different choices for the audio channel or channels on which you'd like SAY to speak. You specify the channels by using an integer from 0 to 11 to designate one of the following channel assignments:

0	=	Channel 0
1	=	Channel 1
2	=	Channel 2
3	=	Channel 3
4	=	Channels 0 and 1
5	=	Channels 0 and 2
6	=	Channels 3 and 1
7	=	Channels 3 and 2
8	=	Any available channel that comes out of the left audio port
9	=	Any available channel that comes out of the right audio port
10	=	Any available pair of channels that come out of both the right and left audio ports (the default)
11	=	Any available single channel

Choices 0 to 3 work just like the channel assignment in the SOUND statement. Choices 4 to 7 are pairs of channels that use both the left and right audio ports so you can hear the voice from both ports. Choices 8 and 9 let you specify the left or right audio port, but lets BASIC choose which channel to use. This is very useful if you're using SAY while the BEEP or SOUND statements are also producing sounds over the audio channels, because SAY can use whatever channel is left over to speak.

Choice 10, the default channel-assignment choice, specifies any pair of channels that are not being used by other sound statements, and that use both the left and right audio ports. In other words, SAY will use any of the channel pairs listed in choices 4 through 7. Finally, choice 11 specifies any available channel not being used by other sound statements. It's probably the safest

channel assignment choice to make, since it's always more likely that a single audio channel will be available than it is that a left-right pair of audio channels will be open for use. However, if you use choice 11 and you have the Amiga hooked up to a stereo system with two speakers, you won't be able to predict which speaker the voice will come out of.

Choosing synchronous or asynchronous speech

Element number 7 of the array is either 0 or 1. The default, 0, specifies synchronous speech. A 1 specifies asynchronous speech. If you choose synchronous speech, BASIC will wait until each SAY statement is finished speaking before it moves on to the next statement. If you choose asynchronous speech, BASIC executes a SAY statement and starts it speaking, then executes the next statement immediately while SAY finishes speaking on its own.

Asynchronous speech speeds up a program because the program can work on other statements at the same time as it speaks. However, other sound-producing statements (like SOUND and BEEP) that follow an asynchronous SAY statement can foul up SAY's speech by starting before SAY is finished speaking. Synchronous speech keeps conflicts like that from happening.

Choosing multiple SAY options

Element number 8, the last element of the array, specifies the way Amiga BASIC will execute multiple SAY statements if they use asynchronous speech. This element has no effect if the SAY statements use synchronous speech (that is, if element 7 is set to zero). There are three choices:

Value	Meaning
0	If BASIC encounters a second SAY statement before the first is finished speaking, it will wait until the first SAY statement is finished speaking before it starts the second SAY statement. Zero is the default mode.
1	If BASIC encounters a second SAY statement before the first is finished speaking, it stops all speech entirely so the first statement is cut off in the middle of speech and the second statement doesn't speak at all.
2	If BASIC encounters a second SAY statement before the first is finished speaking, it stops the first statement in the middle of its speech and starts the second statement's speech immediately.

The purpose of these choices is to avoid garbling by preventing more than one voice from speaking at the same time.

SAY STATEMENT EXAMPLES

It's simple to use a specification array to change the quality of SAY's speech. You don't need to dimension the array, since it's less than 11 elements long, and you can use a READ statement in a FOR...NEXT loop to read the values from a DATA statement for each element. For example, the following short program sets the specifications for the SAY statement to speak in a rapid female voice. It stores them in the array *voice%*, and uses it in a SAY statement that speaks the tongue twister "Peter Piper picked a peck of pickled peppers":

```
FOR i = 0 TO 8
  READ voice%(i)
NEXT i
x$ = "Peter Piper picked a peck of pickled peppers."
SAY TRANSLATE$(x$), voice%
DATA 250, 0, 400, 1, 24000, 64, 10, 0, 0
```

When you run the program, you may notice that it pauses briefly to load something from the BASIC disk. This is because BASIC needs to open the speech device when it first encounters a SAY statement in a program. Any SAY statements that follow will be spoken immediately—BASIC needs to open the speech device only once each time the program is run.

If you want to play around with different voice qualities, you can change the values in the DATA statement, then run the program once again.

The next program reads four different voice-specification arrays and uses the values to create four different SAY voices. The first voice is a female voice, created by raising the base pitch of the default voice from 110 to 220 cps. The second voice is a laconic-sounding robot voice created by lowering the base pitch to 70 cps, choosing monotone, and setting the speech rate to a slow 80 words per minute. The third voice has a nasal sound created by raising the base pitch to 320 cps and lowering the sampling frequency to 10000 cps. The last voice is very rich and deep, but fast; the base pitch is dropped to 65, the speech rate is sped up to 350 words per minute, and the sampling frequency is raised to 28000.

```

FOR i = 0 TO 7
  READ voice1%(i), voice2%(i), voice3%(i), voice4%(i)
NEXT i
DATA 220, 70, 320, 65
DATA 0, 1, 0, 0
DATA 150, 80, 110, 350
DATA 1, 0, 0, 1
DATA 22200, 22200, 10000, 28000
DATA 64, 64, 64, 64
DATA 10, 10, 10, 10
DATA 1, 1, 1, 1
SAY TRANSLATE$("This is voice 1 speaking to you now."), voice1%
PRINT "This is voice 1 speaking to you now."
SAY TRANSLATE$("This is voice 2 speaking to you now."), voice2%
PRINT "This is voice 2 speaking to you now."
SAY TRANSLATE$("This is voice 3 speaking to you now."), voice3%
PRINT "This is voice 3 speaking to you now."
SAY TRANSLATE$("This is voice 4 speaking to you now."), voice4%
PRINT "This is voice 4 speaking to you now."
FOR x = 1 TO 3000: NEXT x

```

When you run the program, you'll notice that even though the PRINT statements follow the SAY statement they describe, they print on the screen at the same time the SAY statement starts speaking. That's because all four voices are in asynchronous speech and BASIC proceeds immediately to the PRINT statement as soon as the SAY statement starts speaking. The FOR . . .NEXT loop at the end of the program is a timing loop to prevent the List window from appearing before voice 4 is finished speaking.

If you change the second to the last DATA statement to a line of four zeros, the PRINT statement will wait until each voice is finished speaking, because the SAY statements now use synchronous speech. (If you switch to synchronous speech, the timing loop at the end of the program is no longer necessary.)

Now that you've had a chance to try out the Amiga BASIC sound statements, you can use them for your own creations. BEEP and SOUND will create sound effects and notes, WAVE will create timbres, and SAY and TRANSLATE\$() will control the Amiga's speech. You can synchronize your SOUND creations with SOUND WAIT and SOUND RESUME. By tying all of these statements together, you can create music, distinctive voices, and other audio effects that will really enhance your programs. And if you enjoy the dynamic quality of your sonic creations, you'll also enjoy reading about the next section's subject: animation.

SECTION 4

This section introduces you to animation on the Amiga. Chapter 10 teaches you the fundamental concepts of computer animation. Chapter 11 shows you how to use Deluxe Video to create your own video animation, and introduces you to other hardware and software products that will help bring your animation ideas to life. The two chapters that end the section (Chapters 12 and 13) discuss the Amiga BASIC animation commands, and show you how to combine them to animate figures in your BASIC programs.

Animation





**CHAPTER TEN
A COMPUTER
ANIMATION PRIMER**

Today's world abounds with animation, thanks to the ubiquitous medium of television. From classic hand-drawn cartoons like Bugs Bunny to computer-generated animation hawking blue jeans, we see images that previously existed only in the animator's imagination. Animation gives life to scenes that are impossible to photograph in the real world.

Traditional animation has never been easy to create. It takes a lot of skill and effort on the part of animators to create the huge number of drawings necessary for even a few minutes of animation, and careful work by technicians to transfer those drawings to film. Animation has seldom been the product of a single person—it's usually created by teams of artists sharing arduous tasks to create a finished product.

Computers are changing animation dramatically. With their power to create images simply and quickly on a monitor screen, computers can take over much of the tedious work animators usually do, drawing backgrounds and moving objects with uncanny consistency and accuracy. Since computers can store images in memory, the animation need never be transferred to film—it can be stored on disk for later playback. Most important, a computer can be an animation tool powerful enough to allow an individual to turn a personal vision into animated scenes of surprising complexity. Using your Amiga, that individual animation artist can be you.

FUNDAMENTALS OF ANIMATION

If you've never done any animation before, you should know some of the fundamentals so that you can understand animation's different elements and how to handle each of them. In the previous primers of this book, you learned about light and color for graphics and about the qualities of sound for music. Now, in this chapter, you'll learn about motion and perspective, two very important aspects of animation.

THE ILLUSION OF MOTION

Animated motion is an illusion. What you see as a continuous movement of objects on a screen is in reality a rapid succession of still images. Film animation presents 25 images per second to the

eye; video animation can present 60 images per second. Why don't you see each of those images individually, like a very speedy slide show? There are two answers: continuity and presentation speed.

Your mind is more likely to accept a series of still images as one single image moving over time if the images are all similar, especially if there is little change in position between images. This establishes continuity from image to image. Figure 10-1 shows six images of a bird flapping its wings. If you see the six images in rapid succession, centered in the same spot, your mind will probably interpret them as one bird flapping its wings. This is true even if the images are shown in slow succession. The human mind likes to bind together separate elements into a meaningful whole.

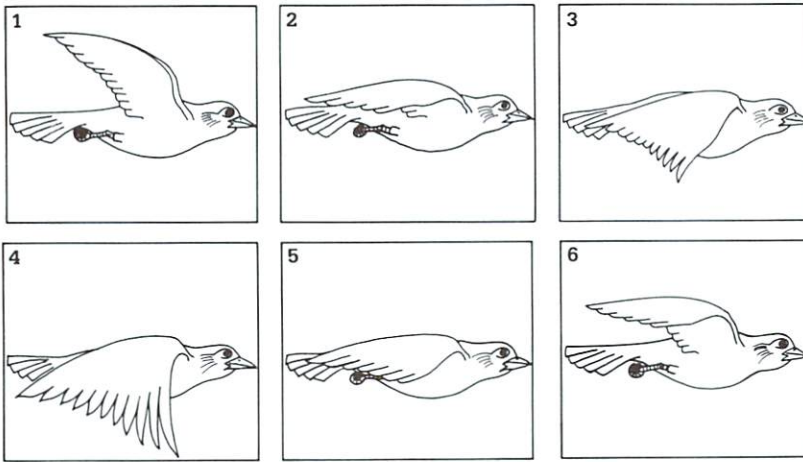


Figure 10-1.

Six successive images of a bird flapping its wings.

A rapid presentation speed helps maintain the illusion of motion. If you show the six images in Figure 10-1 at 6 images per second, you'll see a single bird flapping its wings, but the motion will look jerky. In the back of your mind, you'll know that the motion is actually many individual images. If you show the same six images at 12 images per second (so the bird flaps its wings twice as fast), you won't notice nearly as much jerkiness. The motion looks smoother and more continuous. Once you reach a speed of 25 images per second (the standard speed of movie film), your eyes and mind are tricked into seeing smooth motion. Of course, if there is no continuity from image to image, 25 images per second won't sustain the illusion of motion. For example, if you showed the slides of your summer vacation at 25 images per second, you would see a meaningless jumble. Each slide would have a totally different content, so there would be no continuity.

If it takes at least 25 different images (usually called frames) per second of animated motion, you can see that even a one-minute animated commercial needs 1500 different frames. That's a lot of work to do if you have to create each frame by hand. It's even a lot of work if you create the frames using a computer. Fortunately, there are shortcuts to use so that you won't have to draw 1500 individual frames.

One such shortcut is a common animation technique that involves creating each new image in a sequence of images by copying the preceding image and altering only the parts of the image that are in motion. Because the stationary elements don't change at all from frame to frame, you can separate the moving elements from the stationary elements and concentrate on redrawing just the moving elements.

The simplest way to separate moving elements from stationary elements is to separate them into two parts: the background and the moving objects. The background can be something like a wall, a mountain, or a forest that doesn't move, but provides a backdrop for the action of moving objects. Moving objects can be people, animals, airplanes, or anything else you want to animate.

The background

A good background sets the locale and mood for animation, like the scenery in live theater does for a play. Since you might only have to draw the background once for an animated sequence, you can afford to spend a lot of time to make it rich in detail and a useful backdrop for the animated objects that will move around in front of it.

Although a background usually remains stationary during an animated sequence, there are times when it too will move, usually to indicate a change in viewpoint. For example, put your hand in front of your face and look at your fingers. Notice the background behind your hand. Now turn your head and hand slowly to the right as you continue to watch your fingers. Notice that the background seems to be moving to the left in relationship to your fingers. By moving the background slowly to the left or right, up or down, or diagonally, you can make it appear that the viewpoint is changing in your animation. In traditional animation, this action is called panning. In computer animation, it's called scrolling.

The background can also change if the viewpoint moves closer or further from the background. The background gets larger as the viewpoint moves toward it, smaller as the viewpoint moves away from it. By steadily enlarging the background, you can make it appear that the viewer is moving forward. Likewise, by steadily shrinking the background, you can make it appear that the viewer is moving backward.

Traditional animators create the background by painting it and putting it on an animation stand, a flat surface similar to a drafting table with an attached movie camera set up to photograph one frame of movie film at a time. The animator can use the animation stand to move the background by moving the painting slightly from frame to frame (panning the background), or by moving the camera closer or farther away from the background from frame to frame (moving the viewpoint in or out). This is a time-consuming process, because each sequence of background movement requires the animator to move either the background painting or the camera just the right amount, expose one frame of film, move the painting or camera again, and repeat the process again and again.

Moving the background is much easier for the computer animator; he can create the background painting using the computer's drawing software, and then store it in the computer's memory. Once the drawing is in memory, the animator simply instructs the computer to pan the background by scrolling the image in the desired direction, or to move the viewpoint in and out by shrinking or enlarging the image through manipulation of the picture data.

Moving objects

Once the background is in place, the animator creates objects to move around in front of it. Each of these moving objects can have two basic kinds of motion: external and internal. External motion is the movement of the entire object in relation to the background. Internal motion is the movement of the parts of an object. Consider the example of a man walking across a street. His external motion is his entire body crossing the street. His internal motion is his legs striding back and forth, his arms swinging, his torso bobbing up and down, and a grin passing over his face.

In traditional animation, the animator creates moving objects by drawing or painting them on clear sheets of acetate (called gels). These gels are placed on top of the background painting on the animation stand for filming. Since the acetate allows the background to show through wherever an object isn't painted, the movie camera photographs the gels and the background together as a frame that looks like a single completed drawing, even though there may be several layers of gels on top of the background painting.

To create external motion alone, the traditional animator simply moves a single gel small distances over the background as he shoots successive frames. A computer animator uses a similar process: He first draws the object, and then instructs the computer to display it in front of the background. He then gives the computer instructions telling it where to move the object. Both traditional and computerized methods for producing external motion

are fairly simple. Unfortunately, external motion without internal motion is usually very unconvincing, and looks frozen and cheap. If you try to animate a man crossing the street without moving his arms and legs, it looks like a lifeless paper doll being pushed across the screen. External motion alone might work for something as simple as a rock dropping from a cliff, but even a falling rock would look much more realistic if it tumbled and turned as it fell.

Creating internal motion often takes more work than any other part of animation, which is only fair when you consider that it adds the most life to animation. To create internal motion, the animator has to draw a series of successive images to show how the object changes. The six drawings of a bird flapping its wings (used earlier in the chapter) are an example of creating internal motion through successive images. To show half a minute of this bird flying, you would have to create hundreds of successive wing-flapping, head-twisting, and body-turning images to capture all the variety of motions that a real bird makes in flight.

To create internal motion, a traditional animator draws a sequence of gels, each showing a slightly different position of a moving object. By photographing the gels in succession on top of the background painting, the animator creates animation with internal motion. A computer animator can use a similar method; he creates a series of images in computer memory that the computer displays in sequence on the screen. Neither the traditional nor computerized method for creating internal motion is easy—each requires a lot of time drawing images.

You can take advantage of the cyclic nature of movement to make the task of creating internal motion much simpler. For example, birds flap their wings down, then up, then down, then up, and so on as they fly. If you ignore some of the minor variations that occur from wingflap to wingflap, you can draw a sequence of images that animate just one wingflap, and then repeat that sequence over and over for each flap of the bird's wings—a process called cycling. Instead of creating hundreds of images for a half minute of bird flight, you can create just a few images that you can repeat over and over again as the bird flies. Cycling a sequence isn't as convincing as drawing all the images individually (as you can see in cheap Saturday morning cartoons, which use a lot of cycling), but it does give you internal motion with a fraction of the work it would normally take.

Once you create a series of images to internally animate an object, it's important to make sure that internal motion matches external motion. For example, if you create a man who walks

swinging both arms and legs and don't give him some external motion (or at least move the background behind him), he'll look like he's slipping on ice, not getting anywhere. Likewise, if the man has too much external motion, he'll look like he's wearing seven-league boots, taking giant strides with each small step.

ADDING PERSPECTIVE

Animation is ultimately displayed on a two-dimensional medium such as a computer monitor or a movie screen. You can use different animation and drawing techniques to transcend the two dimensions of the screen and portray your animated images in any of three different perspectives: two dimensions (2-D), three dimensions (3-D), and two and a half dimensions (2½-D). Each perspective has its advantages and disadvantages.

2-D perspective

A two-dimensional perspective assumes there is no depth to the picture, and that the objects in motion move horizontally and vertically, but not toward and away from the viewer. 2-D perspective usually works best for animation that is a symbolic representation of reality (not attempting to look real).

A good example of 2-D animation is cars moving on a map of city streets, as seen below in Figure 10-2. Since cars don't as a rule fly away from the streets, the cars on the map won't move toward or away from the viewer. Their motion is strictly horizontal, vertical, or diagonal—all two-dimensional motion.

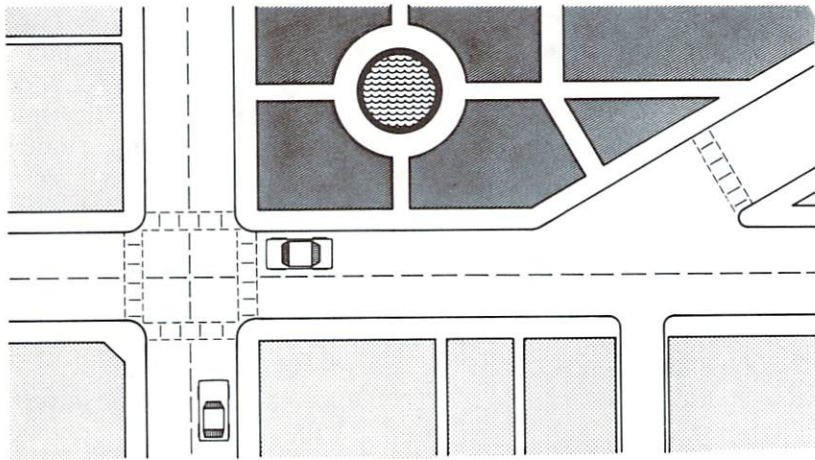


Figure 10-2.

Simple 2-D animation:
cars on a map.

One important property of 2-D animation is that a moving object can't pass in front of or behind other moving objects (at least not if you keep it realistic). Since the objects are confined to a two-dimensional plane, they collide when they meet. For example, two cars in the map example can't pass through each other. One has to go around the other, or they will crash.

2-D animation may not look realistic, but it has some advantages over other perspectives. First, it's usually much simpler to create than 2½-D and 3-D animation because you don't have to keep track of an object's depth, only of its location on a single two-dimensional plane. Second, a simple symbolic representation can show the important elements of the animated sequence with much more clarity. If the streets in the map example were drawn as they would actually look from the air with hills and valleys and black pavement roads, it would be a much more realistic animation, but it would also be much harder to see what was happening as the cars moved around the streets.

3-D perspective

A three-dimensional perspective adds depth and realism to the image on the screen. Objects in 3-D animation can move toward and away from the viewer as well as horizontally and vertically. Since the third dimension, depth, is portrayed on a two-dimensional screen, it's strictly an illusion sustained by different animation techniques.

The most familiar technique used to create a 3-D perspective is the artificial horizon, an invisible horizontal line used by animators as a point of reference. If you're not familiar with artificial horizons, consider a real horizon. When you look far out over the ocean, the line where the sea meets the sky is the horizon. Any boats floating in the water close to you appear far below the horizon. As they sail away from you, they seem to rise closer and closer to the horizon as they get smaller and smaller and then finally disappear. Likewise, any airplanes flying from directly above you to far out over the ocean at first appear quite a distance above the horizon, then drop closer and closer to the horizon and get smaller and smaller until they disappear.

Figure 10-3 shows two ships sailing toward the horizon. Since the one in the lower left corner is larger and further from the horizon than the one in the middle, it appears closer to the viewer.

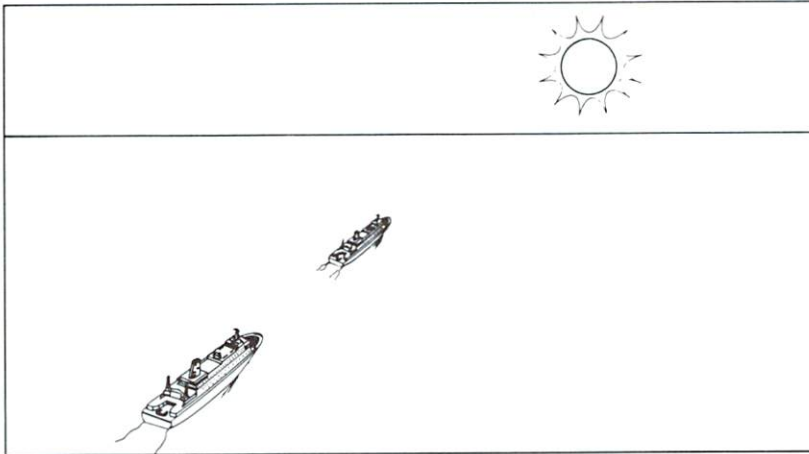


Figure 10-3.

Two ships sailing toward the horizon.

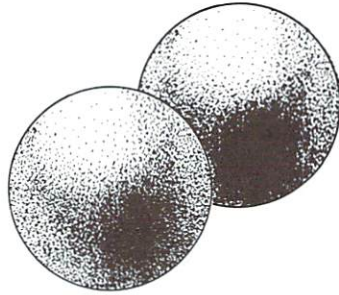
An artificial horizon works the same way. It usually runs across the center of the screen, and may not be visible, since it might be covered by buildings, trees, mountains, or other objects. Even if it's invisible, you can use its location as a reference to make moving objects appear to move toward or away from the viewer. To make an object appear to move away from the viewer, you shrink it as you move it from its original position to the artificial horizon. You can shrink the object so much that it disappears, making it seem like it has moved too far away to see. (The point at which it disappears from sight is called the vanishing point.) To make an object appear to move toward the viewer, you enlarge it and move it out from the artificial horizon. The faster you shrink or enlarge the object, the faster it seems to move away from or toward the viewer. You can combine shrinking and enlarging with vertical and horizontal motion to make an object seem to move in any direction consistent with the three-dimensional perspective of the animation.

To enhance the feeling of depth, you can also change the color of an object as it moves toward and away from the viewer. Objects at a distance are usually dimmer and paler than objects close up. Dimming a moving object as it recedes will add to the feeling of depth. Another aid to developing a 3-D perspective is to diminish internal motion as an object moves away from the viewer. For example, the arm motions of a waving woman standing close to you are very noticeable. If the same waving woman was across a football field from you, the motion of her arms wouldn't be nearly as noticeable.

One of the easiest ways to establish the depth location of a moving object is to pass it in front of or behind another object on screen. If the object passes in front of another object, as it does in Figure 10-4, it's obviously closer to the viewer than the second object. If it passes behind another object, it must be further away from the viewer. If two objects collide and don't pass over or behind each other, then both objects are the same distance away from the viewer.

Figure 10-4.

Two overlapping figures. The figure on top appears closer than the figure underneath.



When you use 3-D perspective techniques, it's important to keep them consistent with each other. If you make an object recede in the distance by shrinking it and then you pass it in front of an object that seems to be close to the viewer, it will look very confusing, and the illusion of three dimensions can be shattered. You must keep track of the relative location of all moving objects to ensure that they pass over or behind the other objects as they would if you were viewing real objects in a three-dimensional space.

The advantages of a 3-D perspective are obvious—it's very realistic, and it draws the viewer into the picture in a way other perspectives can't. The disadvantage is also obvious—it takes a lot more effort, even if the computer is doing the dirty work!

2½-D perspective

A two-and-a-half-dimensional perspective is a compromise between the simple 2-D perspective and the complex but more realistic 3-D perspective. Objects in 2½-D animation are restricted to horizontal and vertical motion in a single plane as they are in 2-D animation, but there are several planes in the picture to create an illusion of depth.

To get a good idea of how 2½-D perspective works, think of a theater stage with an underwater mountain painted on a flat backdrop. This is one plane of the image the audience sees from the theater seats. In front of the backdrop is an aquarium, full of fish, wide enough to stretch to both sides of the theater and tall enough to extend to the top of the proscenium, but only four inches from front to back. This is a second plane of images. In front of the aquarium is a second aquarium just like the first one, also full of fish, which creates the third plane of images. In front of both aquariums and the backdrop is a facade of coral that rises halfway up the height of the aquariums. This is the fourth plane of images. Figure 10-5 shows you how these planes in the fictitious theater are set up.

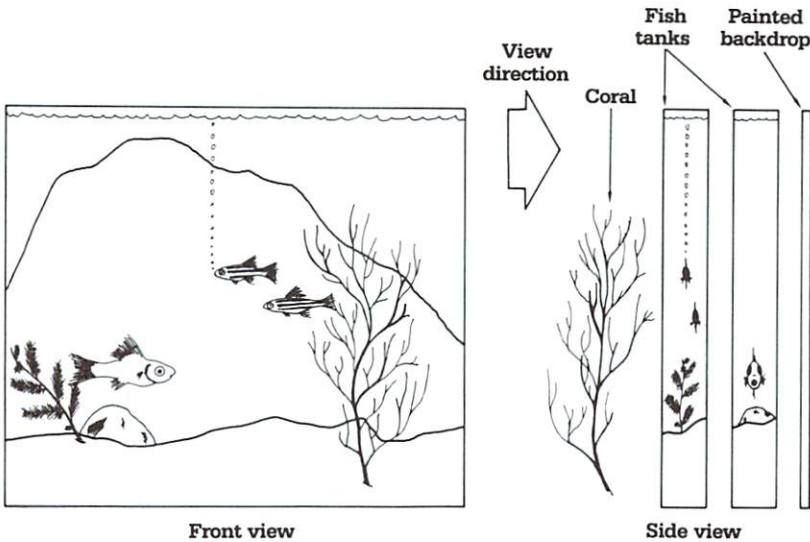


Figure 10-5.
2½-D perspective demonstrated using theater props and two absurdly large fish tanks.

When you look at everything on stage from the audience, you see coral in front. Behind the coral, you see fish swimming back and forth, some passing in front of other fish. Behind the fish you see an underwater mountain. Even though there are only four planes of depth, you can see it as a full-depth picture.

If you were the fish director, you would never have to worry about the fish in the back tank upstaging the fish in the front tank, because they can only move horizontally and vertically, and can't swim out of their tank into the tank in front of them. Likewise, as a 2½-D animator, you don't have to worry about objects moving

toward or away from the viewer. They stay in their plane of motion, passing behind objects in planes before them, passing in front of objects in planes behind them, and colliding with objects in the same plane.

2½-D animation is simpler than 3-D animation. You don't have as much to worry about in terms of changing the size, color, and amount of internal motion of an object, and keeping track of which object passes in front of another object becomes much easier.

INTERACTIVE ANIMATION

In traditional animation, the animator draws the elements of each image, assembles them on the animation stand, and records each increment of movement on individual frames of film to be played back later in sequence. When you play back the film, you see just what was recorded; the animation doesn't change from playback to playback unless the film wears out or breaks.

Computer animation has an advantage over traditional animation: The computer draws each image and places it on the screen according to your instructions. As a result, you can quickly change the way the animation proceeds, using a controller like a mouse or a joystick to tell the computer to change the way the objects move, what part of the background you see, and other aspects of the animation. This is called interactive animation, since it immediately responds to your wishes, and you can respond to its moving images on the screen.

The most common example of interactive animation is the pointer on your Amiga screen. Every time you roll the mouse, the Amiga moves the pointer to a new location on the screen. You can respond to the new location by clicking a mouse button if the pointer rests on an icon, or by rolling the mouse again to move the pointer to the location you want.

Another common example of interactive animation is a video game, which is often a showcase for a computer's animation ability. When you press the joystick button to blow up an invading eggplant, the computer responds to the button push with an animated sequence of the eggplant turning into an exploding fireball. Game players often respond to the animation with increased pulse, sweat on the forehead, and a tendency to push the joystick to its breaking limit.

Interactive animation is one of the computer's greatest artistic capabilities. It draws the viewer into the action more completely than any form of passive animation.

ANIMATION ON THE AMIGA

The Amiga was designed specifically to have superior animation powers. The custom chips can move and manipulate large blocks of graphics data with great speed, a necessary requirement for computer animation. The system software includes libraries and devices that take full advantage of the hardware graphics abilities and make them easily available to programmers. As you work with different animation applications and develop your own Amiga BASIC programs, you'll have a chance to use many of the special features described here.

THE PLAYFIELD

You're probably already familiar with the concept of a playfield on the Amiga, but not with its name. The playfield is the graphic background for any animation. It's usually motionless, and provides a backdrop for the objects that move over it. When you create still pictures with graphics applications like Graphicraft or Deluxe Paint, you're actually drawing figures in a playfield. The cursor or the paintbrush that you move around over the playfield is a moving object, animated by the graphics program and the movements of the mouse.

You'll recall from Chapter 2 that the Amiga can scroll the contents of a window (a playfield) horizontally and vertically. This is a useful tool for advanced programmers to pan the background in animation. For example, as a background for animation, a programmer can create a panoramic view of a mountain scene that is much wider than the window it appears in, as seen below in Figure 10-6. To move the viewpoint to the left, he can scroll the entire playfield to the right. To move the viewpoint right, he can scroll the playfield left.

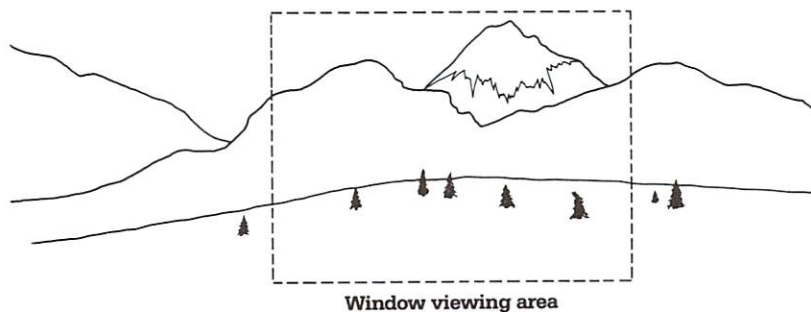


Figure 10-6.

A panoramic background shows through an Amiga window. Scrolling left and right moves the viewpoint.

You'll also recall from Chapter 2 that the Amiga can lay one screen over another with transparent areas in the front screen so the back screen shows through. This gives advanced programmers two different playfields to work with, which is a very handy tool for 2½-D animation. Like the underwater mountain backdrop in the 2½-D animation example, the back playfield provides a backdrop for moving objects in front of it. The front playfield can act as a foreground, much like the coral facade. A programmer can pass moving objects behind the front playfield, but the objects will move around in front of the back playfield. When a programmer scrolls both playfields, he can scroll the front playfield faster than the back playfield to make the background look further in the distance and make the whole scene look more realistic.

GELS

The Amiga allows you to create moving objects in Amiga BASIC or in application programs like Deluxe Video or Aegis Animator. These objects, called gels, can be moved over the playfield. Gel is an Amiga animation term that stands for Graphic ELEMENT, and shouldn't be confused with the gels used in traditional animation. There are two kinds of Amiga gels: sprites and bobs. Sprites are usually small, simple, and fast, and take very little computing work to move around the playfield. Bobs can be much larger than sprites and have more color and detail, but they take more memory to store and much more computing work to move, so they are sometimes slower than sprites. If you understand how the Amiga creates sprites and bobs, you'll understand why they have different characteristics.

Sprites

A sprite is a small object placed directly on the playfield by the Denise chip. It's like a flashlight pointer used on a projected picture: As the flashlight moves up, down, and around, the beam of light moves around without changing the image underneath at all. Like the flashlight beam, a sprite isn't part of the picture drawn on the playfield; as it moves around the screen, the sprite passes over the playfield without changing it.

Sprites are common animation devices on home computers such as the Atari 800 and the Commodore 64. What sets the Amiga's sprites apart are their number and their coloring. You can create up to eight sprites on the Amiga, and each of those sprites can be colored with up to four different colors. One of those colors is "transparent"; you can see the background through transparent parts of sprites. To create a sprite with more colors, advanced

programmers can program the Amiga to combine two sprites to create a new sprite that is the same size as a single sprite, but can use sixteen colors.

Sprites are easy to create, and they take very little memory to store. They're quite useful as fast-flying objects in video games, and also work well as pointers. The pointer in the Amiga Workbench is a familiar example of a sprite.

Bobs

Unlike a sprite, which moves over the playfield without changing it, a bob is a gel drawn directly into the playfield. To get an idea of how a bob moves around the screen, think of a traditional animator creating a sequence of frames by drawing on a single sheet of paper. He draws a swamp pond with a duck floating on the right side of the pond, then snaps a picture of it. To make the duck swim to the left side of the pond, the animator erases the entire duck, then redraws it a fraction of an inch to the left of its original position. He fills in the swamp pond to the right where the duck used to be. He repeats this for each frame of the animation until the duck reaches the left side of the pond.

A bob on the playfield is like the duck in the swamp picture. The blitter (the fast-drawing section of the Agnus custom chip) is the animator. It draws, erases, and redraws the bob and the background underneath the bob with lightning speed to make the bob move smoothly across the playfield.

Bobs can use up to 32 colors at once, and can be drawn in any size as long as they fit in the playfield. Since bobs are drawn and redrawn to move around the playfield, they take more computing time to move than sprites, and are a little more difficult to create. Nevertheless, bobs, once created, can usually move as quickly and smoothly as sprites if they're not too large.

Bobs are useful to create animated objects of complex shape and color. A good use of a bob would be as a large object like a car that travels across the playfield. A sprite wouldn't be able to create an object as large or with as much detail. A familiar example of a bob is a custom brush you pick up from the main picture in Deluxe Paint. When you select a section of your picture, make it a brush, and paint with it, you're actually using a bob.

CREATING INTERNAL MOTION

It's very easy for a programmer to move a gel around a playfield using only external motion. He just tells the Amiga what speed he wants the gel to move, and in what direction it should move. To give the gel internal motion to make it more lifelike, he has two choices: sequenced drawing or component motion.

Sequenced drawing

Sequenced drawing is a method of animation available to advanced programmers using the animation routines in the Amiga's system software, or to dedicated Deluxe Video users. It uses the technique shown earlier in the chapter used to animate the wings of a bird: The programmer draws a sequence of individual pictures of different stages of internal motion. He then plays back the sequence to create motion on the screen.

To use sequenced drawing with a gel, the programmer creates a sequence of shapes and colors, and then uses the Amiga's system software to assign the sequence to a single gel. The Amiga then cycles through the drawing sequence to create an internally animated gel. The programmer can move the sequenced drawing gel around the playfield while it's running through a sequence of drawings just by specifying the distance, speed, and direction for it to move.

As an example, to animate a flying bird, a programmer would produce a cycle of bird shapes that make the bird flap its wings once. He would then assign the cycle of shapes to a gel. The Amiga cycles through those shapes to create a bird flapping its wings. To make the bird fly forward over the playfield, the programmer tells the Amiga how far, in what direction, and how fast to move the gel so the external motion of the bird will match the internal motion of the wing flaps.

Component motion

An advanced programmer can also use component motion, a second method of creating internal motion in a figure. The programmer creates a set of small gels and groups them together using the animation routines in the Amiga's system software to create a single animated figure. Each individual gel has its own external motion (with no internal motion), yet moves correctly with the other gels in the figure to make the set of gels look like one figure with internal motion.

As an example, consider the animated sequence of a man walking across the street. The man has a separate gel for each of his legs, torso, head, and arms. Each of these gels has its own motion: The arms and legs swing backward and forward, the head bobs up and down, and so on. When all these actions combine, the figure of the man goes through walking motions. This is the figure's internal motion. Adding external motion to the figure is done by moving all the gels together in a single direction while they are still going through their walking motion to let the man actually make some progress in walking.

The Amiga's system software takes care of the tricky process of calculating the motion of each individual gel when the entire figure moves in one direction or another. All the programmer has to do is create the individual component gels, define a set of motions for each gel, and then give the Amiga a direction and speed for the external motion of the entire figure.

For short animated sequences, component motion might be harder to create than sequenced drawing. However, it's very useful for long and varied animated sequences, and for interactive animation where the individual gels can change their motion according to the input of the viewer or changing instructions from a program.

GEL PRIORITY AND COLLISIONS

When animating any type of gel on the Amiga, questions arise when two gels collide. Does one gel pass over the other? Do they bounce off each other? Or do they crash and burn? The Amiga uses animation routines in its system software to keep track of each gel's location in the playfield. If any gel touches another gel, or touches the boundaries of the playfield, the system software sends out a signal that identifies which gel collided with another gel or with which boundary the gel collided. The programmer can program the gels to respond: to move away, bend, disappear, explode, pass on as before, or whatever else he has in mind.

In 3-D animation, if one gel passes over another when two gels meet, it's important to make sure the gel that appears closer to the viewer passes over the gel that's further from the viewer. When the programmer creates a gel, he assigns it a priority number. The Amiga's system software keeps track of each gel's priority number. A gel with a higher priority number always passes over a gel with a lower priority number. When the programmer wants to change the depth of a gel, he can change the gel's priority number so that it will pass before and behind the right gels to maintain the illusion of depth.

INTERACTIVE ANIMATION IDEAS FOR THE AMIGA

Interactive animation is easy to create on the Amiga, primarily because of all the different ports the Amiga has to read information from the outside world. The most useful ports are the two controller ports on the right side of the console. You can plug in mice, joysticks, music keyboards, touch tablets, and a variety of other controllers that the Amiga will read. The system software will pass on the controller readings to the animation program so it can change its actions accordingly.

Some intriguing possibilities for interactive animation use fancy hardware. For example, if you add an audio sampler to the Amiga (see Chapter 8 for information on audio samplers) that sends information to an animation program about sounds in the room around the Amiga, the animation program can change the actions of the figures on the screen to match the sounds in the room around them. Likewise, if you have a digital thermometer that sends temperature information to the Amiga through the serial port, an animation program might be able to create different animation for cold weather, cool weather, warm weather, and hot weather.

As you can see, animation on the Amiga is a creative field with a lot of possibilities. To keep the myriad of animation elements under control, you can use an application program like Deluxe Video or Aegis Animator, or you can jump into Amiga BASIC to create your own animation programs. In the next three chapters, you will learn how to do both.



**CHAPTER ELEVEN
AMIGA ANIMATION
TOOLS**

Animation is rarely a spontaneous activity; the typical animator doesn't have a brilliant idea one evening, work a few hours to bring an inspiration to reality, and then perform the work for an adoring public the next day. You, however, can come closer to spontaneity using some of the animation tools available for the Amiga. They make animation a much simpler process.

In this chapter, you'll learn about the available Amiga animation tools. In the first half of the chapter, you'll learn how to use advanced techniques with Deluxe Video, an animation program, to create animated videos with polished results. You'll discover tricks to put more animation on the screen, get smoother results, and spend less time creating a video. You'll also learn how to record your finished videos on videocassette.

In the second half of this chapter, you will read about Aegis Animator, another animation program for the Amiga. You'll also read about the variety of hardware you can add to the Amiga to enhance its video-production capabilities.

MASTERING DELUXE VIDEO

Deluxe Video is an animation program from Electronic Arts that lets you create and play back your own animation sequences (called videos) with accompanying music and sounds. To make animation smooth and uncomplicated, Deluxe Video uses two overlapping playfields for its animation: a background playfield and a foreground playfield. You use the background playfield to create stationary background pictures and the foreground playfield as an area to move objects around the screen. Since the foreground playfield is transparent except where it contains moving objects, the two playfields blend together on your screen so you see the background and foreground together as a single picture, similar to the 2½-D animation example you read about in the last chapter.

Deluxe Video uses three different types of animation. The most common is blitter animation, where bobs are moved around the foreground playfield by the Amiga's blitter, which constantly draws, erases, and redraws them. The second type is sequential animation, where Deluxe Video cycles through a series of individual images you created earlier with Deluxe Paint. The third type of animation is color cycling, the same kind of animation that Deluxe Paint achieves by cycling the color registers.

Deluxe Video is not a simple program; it has to coordinate motion, size changes, file loading (from disk), disappearances, music, sound effects, and many other elements. To use Deluxe Video effectively, you have to compose a video much like a music composer composes music, carefully thinking through the appearance and sequence of each video effect, using the techniques you learn as you gain experience creating scripts. The following sections explain some advanced techniques and tips you can use to create your own Deluxe Video scripts.

USING DELUXE PAINT TO CREATE BACKGROUNDS AND MATCHING OBJECTS

Two of the basic elements you work with in Deluxe Video are pictures that fill the background of your video, and objects that move in the foreground. To create your own pictures and objects, you need to use the graphics-application program Deluxe Paint, because Deluxe Video doesn't actually create custom objects or pictures from scratch; it just loads them from disk and then manipulates them.

To create a background picture on disk for Deluxe Video to use, you simply save a low-resolution Deluxe Paint picture to disk. To create an object for Deluxe Video to use in the foreground, you draw it on a low-resolution Deluxe Paint screen, select it as a custom brush, and then save the brush to disk. Since Deluxe Paint's custom brushes are bobs (as you may recall from Chapter 10), and Deluxe Video uses bobs for its moving objects, Deluxe Video has no trouble loading the brush from disk to use as an object.

When you create objects and backgrounds with Deluxe Paint for a single video, you want them to match in size and style, and you want to make sure that the colors work well together. If you create each object and picture in different Deluxe Paint sessions, you might be disappointed with their consistency when they all come together in your video. You can use some special techniques with Deluxe Paint that will ensure that your pictures and objects work well together in Deluxe Video, and will also save you time.

Maintaining color consistency

The key to keeping colors consistent is to realize how many colors you can use in each of the two playfields that Deluxe Video uses. Pictures appear in the background playfield. They can use up to eight colors, since the Deluxe Video background playfield is three bit planes deep. Objects appear in the foreground playfield. The foreground playfield also uses three bit planes (a different set of three bit planes than the background playfield uses), so you might

think that the objects it contains can also use up to eight colors. Instead, they are limited to seven colors, because the eighth color—the background color—is transparent so the background playfield can show through. And, since the background and foreground playfields can each use different sets of colors, by combining playfields you have a total of 15 colors that you can use in a video.

When you create objects and pictures for a single video script, you can set the Deluxe Paint colors so that you use only one set of eight colors for the background and another set of seven colors for the foreground. Since Deluxe Paint has a spare screen, you can work on the background in one screen, and on the objects for the foreground in the other. Follow the instructions below to create a picture and several objects to use in Deluxe Video.

First, set up Deluxe Paint:

1. Load Deluxe Paint by typing the command **dpaint lo 4**. This loads a version of Deluxe Paint that uses four bit planes and gives you 16 colors to work with. When the Deluxe Paint screen appears, you'll see two columns of eight colors at the bottom of the control strip. You can use the colors in the left column for the background picture and the colors in the right column for the objects.
2. Open the palette so you can change the available colors to your liking.
3. Set the eight colors in the left column to colors you want to use in your background picture.
4. Set the first seven colors in the right column as the colors you want to use in the objects. Since you'll use the eighth object color as the background color, it will be transparent in Deluxe Video, so it doesn't matter what color you choose for the last color in the right column. As you choose the object colors, try to make them different from, but complementary to, the background colors you set in the left column. This will make your video look much more colorful when you put your objects and the picture together.
5. Close the palette by clicking the **OK** button.

Now that Deluxe Paint is set for work, you can create a background picture:

1. Paint your background picture using the colors in the left column of the palette.
2. Choose **Save As ...** from the **Picture** menu to save your background picture to disk. You'll want to save the picture directly on your Deluxe Video Parts disk, so:
3. Insert the Deluxe Video Parts disk in a disk drive.
4. When the **Save** requester appears, click in the **Drawer** box, and use the **DELETE** key to delete the contents of the box. If the Parts disk is in an external drive, first type the name of the drive (such as **df1:**), then type **Pictures**. To enter the background picture name, click in the **File** box, and type the name. Click the **Save** button to save your background picture in the **Pictures** drawer on your Deluxe Video Parts disk.

Once you've created a background picture, you can create objects to go with it:

1. Press **j** to jump to Deluxe Paint's spare screen.
2. Use the right mouse button to select the bottom color in the right column of the palette as the background color, then select the **CLR** button to fill the screen with this color. Since the background color will be transparent on the screen in Deluxe Video, it doesn't really matter what color it is on the palette in Deluxe Paint.
3. Draw your objects in this screen using the first seven colors in the right column of the palette.
4. To check the size and color of any object against the background picture, use the brush-selection tool to select the object as a brush, then switch screens back to the background picture. Use the mouse to move the object around the background picture to see how it looks in motion. When you're finished, switch back to the objects screen.
5. When you're satisfied with the object, use the brush-selection tool to select it again as a brush.

6. Choose the **Save As ...** command from the **Brush** menu to start to save the object.
7. Repeat the steps you used to save the background picture, except save the brush in the **Objects** drawer of your Deluxe Video Parts disk.

If you have other background pictures and objects to create, you can create a new set of colors in the palette, and can use the same techniques to create new background pictures with their own sets of objects. When you're finished creating and saving pictures and objects, quit Deluxe Paint and run the Deluxe Video Maker program. To insert your pictures and objects in Deluxe Video, follow these instructions:

1. Open the scene where you want to insert your background picture.
2. Add a **Picture** track in the scene for your background picture. When a requester appears asking you which picture to load, select the name of the background picture you created earlier.
3. A **Palette More Than 8 Colors** requester will appear. (Although you used only eight colors in your background picture, there were a total of 16 colors in the palette, and Deluxe Video knows it.) Select the **Best** palette option. Deluxe Video will load your background picture, using your original colors.
4. Click the **Select** button in the requester to finish adding the **Picture** track.
5. To load the objects that go with the background picture, add an **Object** track for each object. When the requester appears for each track, select the object you created earlier, select the **Best** palette option to load them using your original colors, then click **Select**.

By using the techniques presented here, you can keep your pictures and objects consistent, and save yourself the trouble of trying to set a color palette in Deluxe Video to match objects and pictures that were created using entirely different palettes.

USING COLOR-CYCLE ANIMATION IN THE BACKGROUND PICTURE

Most of Deluxe Video's animation takes place in the foreground playfield, with moving objects set in motion by **Move To** effect boxes in the script. This type of motion is blitter animation because it moves the object bobs within the foreground playfield. Blitter

animation uses quite a bit of processor time and memory, and makes the motion on screen jerkier as you add more objects. If you want to use a different method of animation that requires very little memory and processor time, and also lets you add animation to the background playfield, you can use color-cycle animation.

You've probably played with color cycling using Deluxe Paint. By choosing a range of colors as a cycle range, painting a succession of those colors on the screen, and then cycling the colors, you can create the illusion of motion. You can also use color cycling in Deluxe Video by choosing which colors will cycle in either the foreground or background playfields. You can use Deluxe Paint to create background pictures or objects to use with Deluxe Video's color-cycling effect.

The following example uses color cycling. It creates a background picture depicting a carnival shooting arcade, with a series of target ducks moving on a conveyor belt at the back of the arcade tent. To create this background picture, first load Deluxe Paint and set the palette:

1. Type **dpaint lo 4** to load Deluxe Paint with four bit planes so you get 16 colors in the palette.
2. Press **p** to open the palette so you can set the colors.
3. The eight colors in the left column are for use in the background picture. Leave the first three colors black, white, and red, but change the fourth to brown. You'll use these four colors to create the carnival tent and a wooden counter top.
4. You'll use the last four background-picture colors to create the ducks. Set the first of the four colors to yellow. Leave the next three colors in their default colors for now—they'll be changed later.
5. Set the last four background-picture colors (the one you just changed to yellow, and the three below it) as color cycle number one (**C1**). (See Chapter 3 for more information if you don't know how to set a color cycle.)
6. Close the palette by selecting **OK**.

Now you can create the carnival tent and ducks:

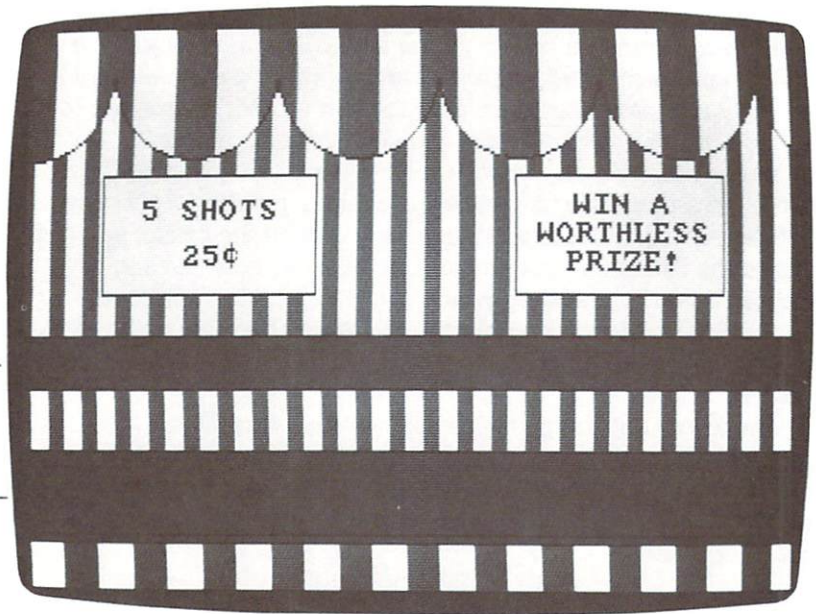
1. Use the first four colors in the left column to draw a carnival tent with countertop and a runway strip for the target ducks as you see in Figure 11-1 on the next page. The brown strip in the middle of the picture is the duck runway.

Figure 11-1.

A carnival tent with a duck runway strip.

Duck runway

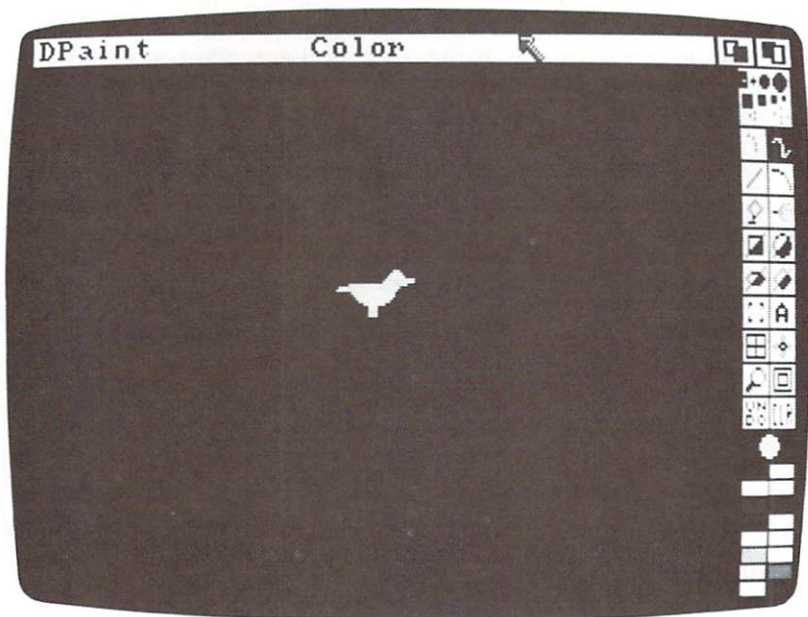
Countertop



2. Switch to the spare screen and draw a single duck in solid yellow like the one you see in Figure 11-2.

Figure 11-2.

A duck to be selected as a brush.



3. Use the brush-selection tool to select the duck as a brush, and switch back to the carnival picture.
4. Choose **Cycle** from the **Mode** menu so the brush will cycle through the colors in the cycle range you set earlier in the palette.
5. Select the first of the four cycle colors in the control strip (yellow) as the foreground color. This makes the brush-color cycle start from this color.
6. Lay down a string of ducks nose to tail, left to right, across the duck runway, clicking the mouse button once for each duck. At each click, the duck should change colors because the brush is color cycling. The end result should look like Figure 11-3.

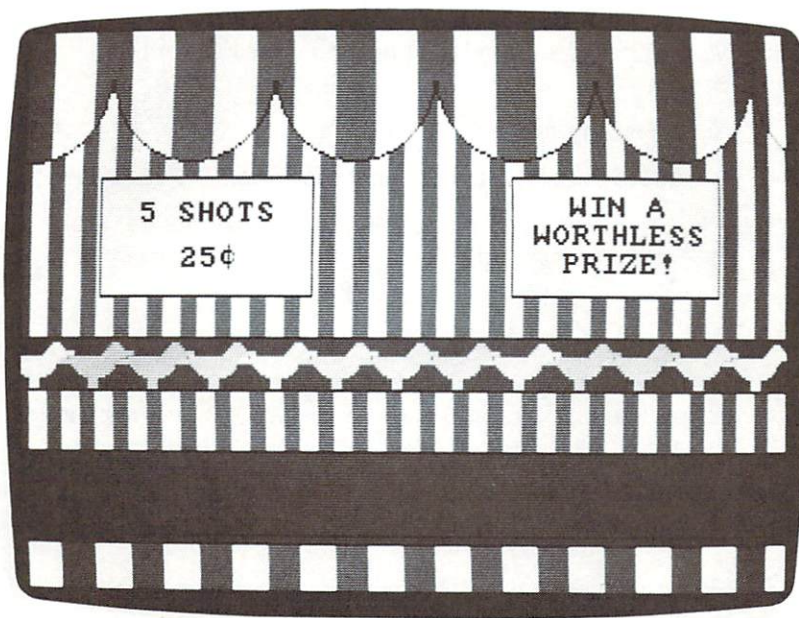


Figure 11-3.

A chain of ducks drawn using colors in the cycle range laid down on the duck runway.

If you cycle the colors by pressing the TAB key at this point, the ducks will look like they're standing still and changing colors. To make them look like they're moving, you have to change the palette once again:

1. Open the palette.
2. Copy the brown that you used to paint the duck runway to the last three colors of the cycle range (colors six, seven, and eight).

3. Select **OK** to close the palette.
4. Press the **TAB** key to test the color cycling. You should see a well-spaced chain of yellow ducks moving across the runway.
5. Save your painting to the **Pictures** drawer of your Deluxe Video Parts disk.

At this point, if you want to create some foreground objects to go with your carnival-tent background such as popcorn, cheap prizes, and BB rifle barrels, you should switch to the spare screen to create and save them as objects in the **Objects** drawer of the Deluxe Video Parts disk. When you're finished, quit Deluxe Paint, run Deluxe Video, and put your cycling picture in a script:

1. Use the techniques described previously to put your carnival picture in a scene using a **Picture** track.
2. Make the picture appear by placing a **Load** effect in the **Picture** track.
3. Add a **Background** track to the scene.
4. Add a **CycleClr** effect to the **Background** track.
5. When the **Cycle Colors** requester appears, make sure colors 4, 5, 6, and 7 are on and colors 0, 1, 2, and 3 are turned off (a color is on if its number is highlighted).
6. Select the **OK** button to close the requester.
7. Drag the first arrow of the **CycleClr** effect to the time in the script where you want the ducks to start moving, and drag the second arrow of the effect to the time where you want the ducks to stop moving. Figure 11-4 shows a scene script with the carnival picture loaded and the color cycling set to last for eighteen seconds.
8. Play the scene to check the speed of the ducks. If they're moving too fast, you can double-click on the top of the **CycleClr** effect to open it again and set the **Speed** slider to get the speed you're looking for.

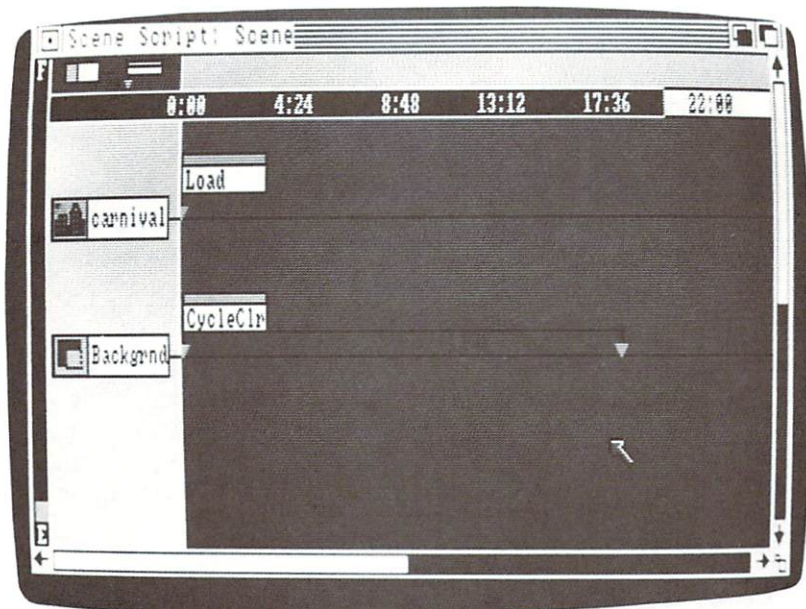


Figure 11-4.

A scene script set up for color-cycle animation on the background playfield.

Once your moving-duck background has been created, you can go on to add moving objects on a **Foreground** track in front of it, and sound effects on a **Sound** track, to create an interesting video.

You can use this color-cycling technique in many ways to make your backgrounds more interesting. You might use it to create flowing rivers, waving trees and grass, moving crowds of people, or any other repetitive background movements. It all runs very smoothly behind whatever action you create with moving objects in the foreground playfield.

CREATING SEQUENCED DRAWINGS FOR ANIMATION

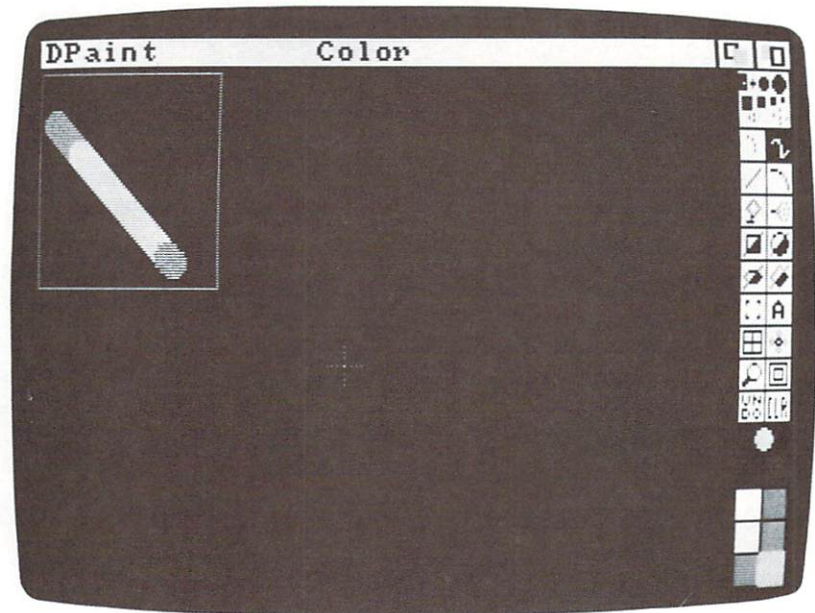
Sequential animation is the most sophisticated animation you can create with Deluxe Video. It's also the most time-consuming animation to produce. To create the set of sequenced drawings Deluxe Video needs for sequential animation, you have to work with three different programs: Deluxe Paint to draw your sequence, the Deluxe Video Framer to combine the sequenced drawings into an object, and the Deluxe Video Maker to make the object appear in a video and to run it through its sequence of drawings. The following set of instructions will help you create a sequentially-animated object with a minimum of work.

The first thing you need to do is draw your image sequence using Deluxe Paint:

1. Run Deluxe Paint using the command **dpaint lo 3** to get a 3-bit-plane version of Deluxe Paint that uses eight colors.
2. Set the last seven colors in the palette to the colors you want to use for your images. You'll use the first color as the background color. It doesn't matter what color you set it to, because Deluxe Video recognizes it as the background color, and makes it transparent in the video.
3. Draw your first image in the upper left corner of the screen, leaving room for the other images in the sequence in the rest of the screen. Be sure to draw your image a little below the title bar. If you draw it too high, you won't be able to frame it later with the Deluxe Video Framer: It will be too high on the screen for the frame to slide up to.
4. Draw a box around the image, as you see in the example in Figure 11-5.

Figure 11-5.

A boxed image ready to be duplicated to create a sequence of images.



5. Use the brush-selection tool to select the box and its contents as a brush.
6. Lay down copies of the box next to each other, as you see in Figure 11-6. Make sure the borders of the boxes don't overlap: Wherever they touch there should be a double-width line. If you run out of room in one row, start a new row going from left to right under the last row.

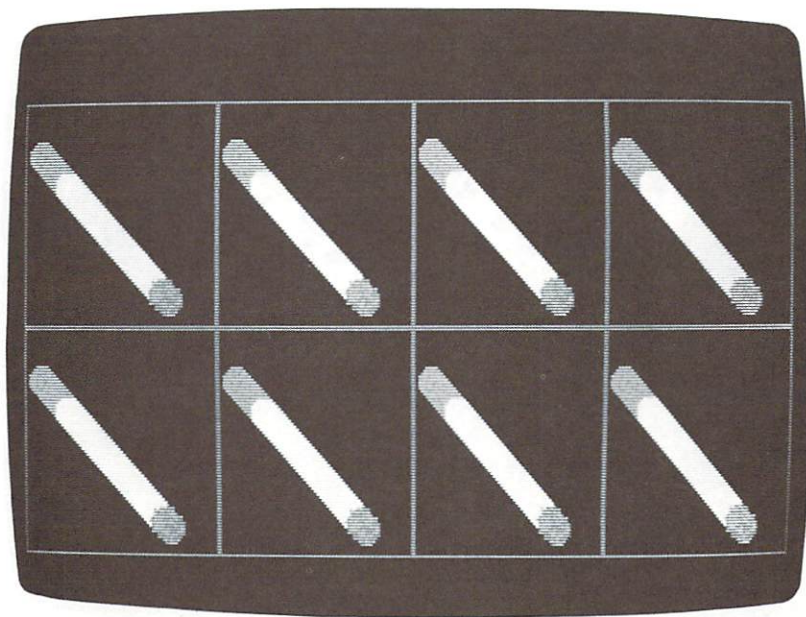


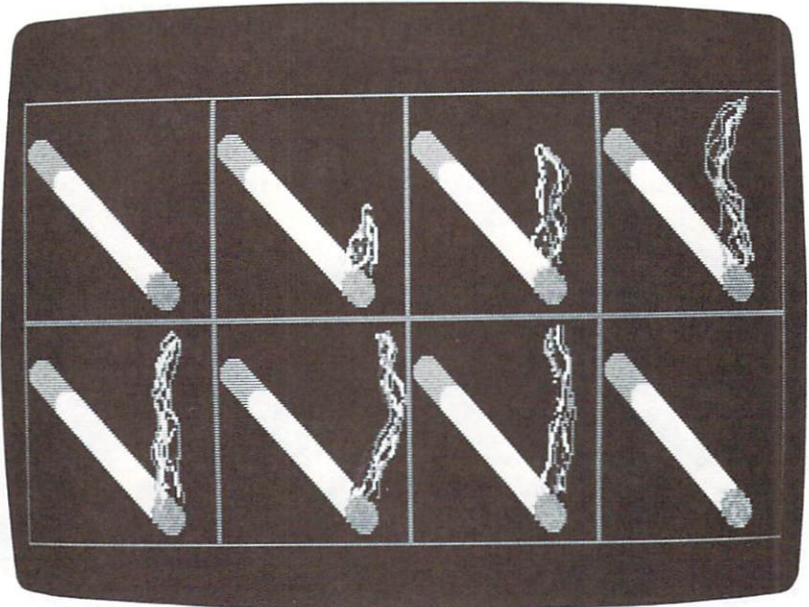
Figure 11-6.

A sequence of images ready to be altered.

7. Alter the images in the boxes so you have a sequence of images that progress just the way you read: from left to right, top to bottom. Remember as you alter the images that the box around the image represents the still frame you'll see the sequenced images through. Use it as a frame of reference to keep the image from moving too much from one frame to the next. Figure 11-7 (on the next page) shows a sequence of changing images.
8. When you're finished drawing, use the **Save As...** command from the **Picture** menu to save the whole sequence as a picture in the **Pictures** drawer of the **Parts** disk.

Figure 11-7.

A sequence of changing images.



Once you've saved the sequence as a picture, you can quit Deluxe Paint and run the Deluxe Video Framer to turn your picture into an animated sequence of images:

1. Run the Framer.
2. Load the picture you just created.
3. Choose **Change Frame...** from the **Project** menu.
4. When the **Frame Sequence** requester appears, set the number in the left half of the requester to match the number of frames across, and set the number in the right half of the requester to match the number of rows of framing in your picture. In this example, the numbers would be 4 and 2, respectively. Select **OK** to exit the requester.
5. Drag the upper left corner of the frame grid that appears until it lines up with the upper left corner of your set of boxes.
6. Drag the lower right corner of the box labeled "a" down and to the right to size the frame grid to exactly cover the borders of your boxes.

- The images will have letter names in the upper left corner of each section of the frame grid like the images in Figure 11-8. Note what happens in each frame—when you use the Deluxe Video Maker later, it lets you change the sequence order by typing in the frame letters in the desired order.

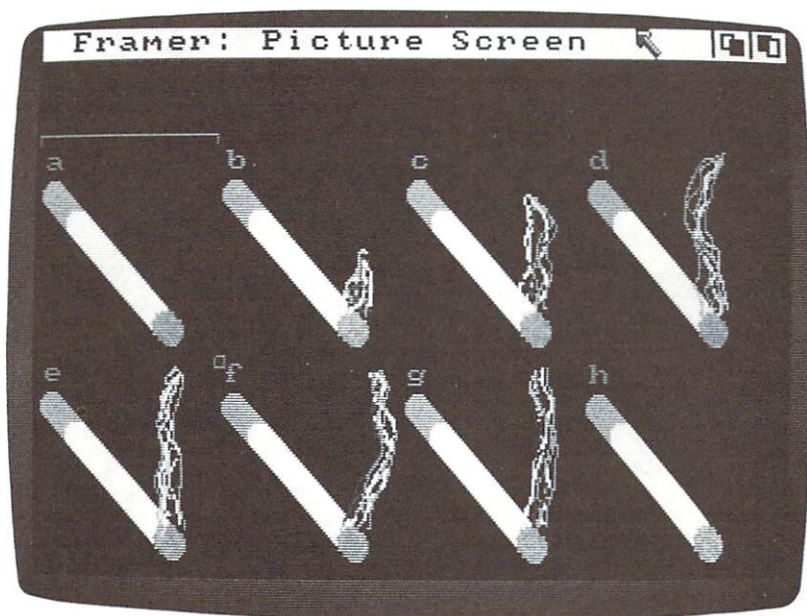


Figure 11-8.

A sequence of images framed and lettered in the Deluxe Video Framer.

- Choose **Make Object From Frame** from the **Project** menu to combine the images into an animation sequence.
- Drag the single frame remaining on the screen to the spot on the screen where you want to see it, then choose **Animate** from the **Project** menu to see the framer run through your sequence of images.
- Choose the **Save...** command from the **Object** menu to save your sequence as an object in the **Objects** drawer of your Parts disk. Note that Framer will add a **-1** extension to your old filename for you—just click **Save** to save the sequence under this name.

Once you've set up your animation sequence with the Framer, you can quit the Framer and use the Deluxe Video Maker program, where you can put your animation sequence in a script:

1. Run the Deluxe Video Maker.
2. Add an **Object** track to a scene.
3. When the **Object** requester appears, choose the name of the animated sequence you just created.
4. Add an **Appear** effect to the **Object** track.
5. When the **Appear Where?** requester appears, position the square where you want the object to appear on the screen and select **OK** to leave the requester.
6. Add an **AnimSeqn** effect to the **Object** track after the **Appear** effect.
7. When the **Animate Sequence** requester appears, it shows the letter names of each picture in your animation sequence. Look at the notes you took when you used the Framer to remember the order that you want the pictures to appear in, then enter the order using letter names in the **Sequence** box, first deleting what's already there if necessary. If you want Deluxe Video to go through several cycles of the sequence you just entered, move the **Repetitions** number up to the number of cycles you want, then select **OK** to leave the requester.
8. Drag the left arrow of the **AnimSeqn** event to the time in the script where you want your animated sequence to start running, and drag the right arrow to where you want the sequence to stop running. Deluxe Video will run through the number of images in the sequence you specified over the time period set for the effect, so if you stretch the start and stop points of the effect out over a long period of time, the animated sequence will step through the images slowly. If you limit the effect to a short period of time, the sequence will cycle through the images quickly.
9. Play the scene to see your sequential animation. If you have the **AnimSeqn** effect stretched out to the time you want and the images are cycling too slow or too fast, then you can open the **AnimSeqn** effect again and add or subtract from the number of images in the sequence, or increase or decrease the number of repetitions.

You can use sequential animation to very good effect. Keep in mind that while the object is cycling through its sequence of images, you can move it with a **Move To** effect. If you create a walking or flying object, you can coordinate its external motion (set with the **Move To** effect) with its internal motion (set with the **AnimSeqn** effect) to make it look as if it's really walking or flying.

SYNCHRONIZING EFFECTS

As you use Deluxe Video, you'll find it very important to synchronize effects in your script so sounds will accompany the right actions, so motion can start at the same time as an object appears, and so other effects that should happen together do happen together. One easy way to synchronize effects is to start them together by lining up the left arrows of the effect boxes. You can read the script time of the arrow at the top of the script as you drag it along the track.

There are times when you may want to set the beginning of one effect to occur at a specific point in the middle of another effect. For example, you may want to add a "pop" sound in the middle of an animated sequence that shows a pop bottle opening. How do you find the correct point in the sequence without resorting to tedious trial and error? You can use the remote control to locate the exact time of the pop bottle opening.

Whenever you run a video or scene with the remote control showing on the screen, the bottom of the remote control displays the script time, running it forward or backward as you play different parts of the script in different directions. To find the script time of a specific effect in a scene, you can stop the scene from playing and then read the time at the bottom of the remote control. To make it easy to find, you can play the scene to approximately where the effect occurs, and then play the scene using the single-step mode.

To use single-step mode, select the button in the remote control with the three dots in it, then use either the forward-play or reverse-play button to go through the script one step at a time. Whenever you find the exact point in the scene that you're looking for, note the time at the bottom of the remote control. When you go back to the script, you can set a new effect at exactly the time you noted by positioning its left arrow, using the time reading that appears at the top of the script.

When you use this timing method, be sure that the **Realtime** option isn't selected in the **Options** menu. If it's selected, the time display at the bottom of the remote control will keep incrementing during disk-loading operations as well as during video effects, so the readout won't have anything to do with the time location of effects in the script.

MAKING YOUR SCRIPTS READABLE

As you create video scripts, the tracks and effects begin to add up to an on-screen jumble, especially if you have many effects running at the same time on one track. If you want to revise your script, it can be very hard to see what you're doing—some effect boxes may completely cover other effect boxes, effect-box arrows may be crammed into one tiny space so you can't see where the effects start and end, and some effect boxes might be placed so far away from their arrows that you can't follow the connecting lines.

To avoid these problems, follow these rules of thumb to make your script easy to read and revise:

1. Keep related tracks next to each other. This makes it easy to synchronize effects between the tracks, and also makes it easy to follow the related tracks through the script. For example, if you want two birds to fly across the screen accompanied by bird squawks, you use three tracks together: two **Object** tracks (one for each bird) and a **Sound** track. Keep these three tracks together in the script so you can see and synchronize the effects between them.
2. If you have several effects occurring simultaneously in a single track, layer the effects so each type of effect has its own height above or below the central track line. This lets you follow the course of each type of effect, and see how they interact with the other types. For example, in an **Object** track you might have several different **AnimSeqn** effects occurring at the same time as **Move To** effects and **Size** effects. To keep everything clear, you can put all the **AnimSeqn** effect boxes very close to the track line. One level above the **AnimSeqn** effect boxes, you can put the **Move To** effect boxes, and one level above that you can put the **Size** effect boxes as shown in Figure 11-9.
3. If effects are crowded close on each other's heels, making them very hard to read, stretch out the time scale of your script using the **Time Scale...** command in the **Options** menu. This gives you much more space to fit your effects into so you don't have to overlap boxes, and so you can see where the effect arrows are located. For example, if you have a scene script set to show 22 seconds at a time in the scene-script window and the effects are crowding each other on individual tracks, then you can set the script to show seven seconds at a time, stretching out the effects in the track to over three times as far apart as they were previously.

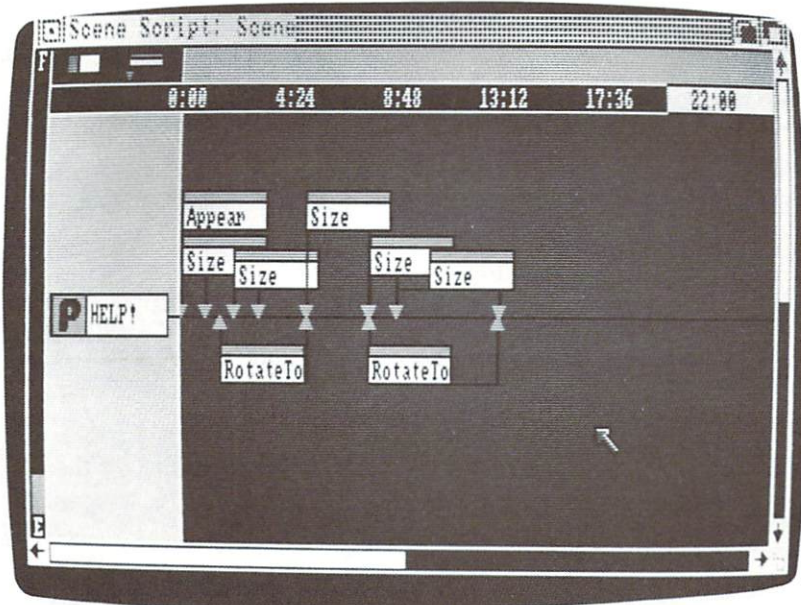


Figure 11-9.

A single Object track with effects grouped by type on separate levels to keep them readable.

RECORDING DELUXE VIDEO ANIMATION ON VIDEOTAPE

Once you've finished creating a video on Deluxe Video, you can record it on videotape. It's very easy. All you have to do is connect the Amiga to a videocassette recorder, start the VCR recording, and then start the animation running. The following sections give you the details you need.

Connecting the Amiga to a VCR

To connect the video output of the Amiga to the video input of the VCR, you need a simple audio cable with a phono plug on each end. Plug one end of the cable into the video port on the back of the Amiga's console. The video port is the port farthest to the right as you look at the back of the Amiga. Plug the other end of the cable into the jack labeled "Video In" (or something similar) located on the back or side of your VCR.

If you have sound in your animation, you should connect the audio ports of the Amiga to the "Audio In" jack (or jacks) of your VCR. If you have a stereo VCR, you connect the Amiga to the left and right "Audio In" jacks of the VCR the same way you connect the Amiga to a stereo cassette recorder. (See Chapter 8 for more details.) If you have a monaural VCR, you connect the Amiga to the VCR the same way you connect the Amiga to your monitor. (See *Introduction to Amiga* in your user manual for more details.)

Making the recording

Once the connections are made, you can start recording. To capture all the video on tape without recording the process of your loading and running the script, follow these instructions:

1. Run the Deluxe Video Maker and load whatever video you want to record. Choose **Play Video** from the **Project** menu.
2. When the remote-control unit appears, stop the video and select the reset (:0) button to set the video back to its start.
3. Click the single-step button on the remote control, then click the play-forward button once to play and freeze the script at its very beginning so the title bar at the top of the screen disappears.
4. Click the single-step button a second time to turn off the single-step mode, then select the back gadget on the remote control to make it disappear.
5. Move the pointer off the bottom right corner of the screen so that no part of it shows on screen.
6. Set the recording speed of your VCR to the highest possible speed for the best video quality, then press the key on the VCR that starts it recording.
7. Wait for a few seconds until the VCR's tape heads are up to speed, then press the Amiga keyboard's right cursor key to start the video playing.
8. At the end of the video, stop the VCR. You've just recorded your video onto tape.

VIDEO RECORDING QUALITY

When you watch your video recording, don't expect the same quality you see when you watch the original video on an RGB monitor. Recording the video downgrades its quality because of two processes: First the video signal is converted to a composite video signal instead of an RGB signal so it can be fed to the VCR; then the composite signal is recorded on videotape, which further deteriorates the quality. There's nothing you can do about having

to use a composite video signal—most VCRs will only accept a composite video signal. You can improve the quality of the tape recording, though, by buying a high-quality home VCR and using a good-quality recording tape. If you buy a Beta VCR, try to get a SuperBeta recorder, which has substantially better video quality than other types of Beta recorders. If you buy a VHS VCR, you should consider an HQ VHS, which also has video quality much improved over other VHS VCRs.

ADDITIONAL ANIMATION SOFTWARE

Deluxe Video is not the only animation software you can buy for your Amiga. Electronic Arts plans some further additions to Deluxe Video, and Aegis Development has an animation program called Aegis Animator.

AEGIS ANIMATOR

Aegis Animator, sold by Aegis Development, is a simple but effective animation program. It's inexpensive, easy to use, and can create some great three-dimensional animation using simple abstract objects. Aegis Development bundles Aegis Animator together with the Aegis Images graphics program, so you can use pictures created in Aegis Images as backgrounds and objects in Aegis Animator videos.

Aegis Animator uses a type of animation called metamorphic animation for most of its effects. In metamorphic animation, the Amiga doesn't calculate the motion of every bit in an object that it's moving around the screen. Instead, it uses simple geometric objects such as lines and polygons that it can define by setting a series of points and connecting them with lines. The Amiga has to calculate only the endpoints of an object as it moves it around the screen instead of calculating the motion of every bit in each object. For example, to keep track of a square, the Amiga has only to keep track of the location of the four corner points. As Animator moves these simple objects around, it calculates new positions for the endpoints and then draws the rest of the object by connecting the points and filling it in if it's a solid object.

The advantage of metamorphic animation is that it takes very little calculation to manipulate objects on screen. Aegis Animator makes good use of this advantage, and offers a variety of special effects you can use on an object, such as moving it from one location to another, changing its size, rotating it around three

different axes, and changing it from one shape to another. Another advantage of metamorphic animation is that the motion of an object is often much smoother than it is in blitter or sequenced drawing animation, because of the reduced calculation time.

The disadvantage of metamorphic animation is that you have to work with simple geometric shapes. This is no drawback if you're working on abstract animation, but it's a distinct disadvantage if you want to create objects like people, animals, and other irregularly shaped items.

Animator offers you some alternatives to metamorphic animation: You can use a 32-color picture created with Aegis Images, Deluxe Paint, or another IFF standard graphics program as a background to your animation. (Animator uses a single 5-bit-plane playfield for animation instead of the two 3-bit-plane playfields Deluxe Video uses, so it's possible to get 32 colors.) You can also create individual objects with up to 32 different colors in a graphics program, and Animator will use them in simple animation; it moves them around the screen, but it won't rotate or size them or change their shapes as it can with geometric objects.

Animator doesn't use a script the way Deluxe Video does. Instead, it lets you create objects on the Animator screen, and then move them around and control them with other effects. You record each step of your motion and control effects, and Animator remembers the whole sequence so it can play it back later. Once you finish a sequence, you can create other sequences, and then tie them all together in a special screen called the storyboard. This makes it easier to create a short video than by using a script, but it also makes it harder to edit a video and to synchronize many effects at once.

Unlike Deluxe Video, Animator is strictly animation—there is no sound, so you can't add sound effects or a musical score to accompany your creations unless you run a music program simultaneously. Also, Animator does not have commands for adding text to your animation.

DELUXE VIDEO LIBRARY DISKS

Electronic Arts plans to follow Deluxe Video with Deluxe Video library disks containing additional pictures, objects, sounds, and scene generators that you can use with Deluxe Video. The contents of the library disks will all be related to give you the tools and parts to easily create business presentations, for example, or to title videotapes as another example. The scene generators will create specialized scenes automatically for you. The library disks will contain the work of professional artists and programmers to make your work easier and better looking.

ANIMATION AND THE IFF STANDARD

Since animation has no set notation standard, and different animation programs use entirely different techniques to create their effects, there is no IFF standard for video scripts. This means that you can't create a video on Deluxe Video and then play it using the Aegis Animator or vice versa. The individual parts of the videos use the IFF standard, though. Pictures, sounds, and music can be transferred from one animation program to another that uses the IFF standard and can make use of them.

ADDITIONAL ANIMATION HARDWARE

You can greatly increase the animation power of your Amiga by adding extra hardware. Some of the hardware will make your animation programs run faster and help store longer sequences of animation than you can with the Amiga alone. Other hardware will add external sources of animation and help you blend the Amiga's animation with external animation.

EXTERNAL RAM CARDS

Animation is an activity that takes a lot of memory. When you consider that a simple video background can take up to 8K of RAM to store, and one sampled sound can take over 30K of RAM, it's no surprise that an Amiga with 512K of RAM can quickly fill up when you create a script with many different backgrounds, dozens of objects, and sampled sounds to match. To get around memory limitations, programs like Deluxe Video store all the animation elements they can in RAM, and keep the rest on a disk in the disk drive. As you play a video, Deluxe Video loads the new elements from the disk when they're needed and erases the old elements from RAM. This is why there are pauses in the video as it plays—Deluxe Video has to take some time to load new animation elements from disk.

You can add up to eight megabytes of additional RAM to the Amiga, using the expansion connector on the side of the console to help store your video elements. This additional memory is called external memory to differentiate it from the 512K of RAM that the Amiga can have internally. The Amiga must keep pictures, objects, and sampled sounds that it works with in the internal memory, since the custom chips that draw, animate, and play sounds have direct access only to the internal memory. (The Amiga's internal memory is sometimes called "chip memory" for this reason.)

Animation elements that the Amiga isn't using at the moment can be stored in external RAM. Since the Amiga's 68000 processor

can directly read both internal and external memory, it can quickly bring the animation elements into the chip memory when they're needed without the delay that would be caused by loading them from a floppy disk.

Several companies make external RAM cards that plug into the Amiga's expansion port. Most of these cards contain two megabytes of RAM, memory enough to store more than the contents of two floppy disks. Some of the RAM cards allow you to add additional 2-megabyte RAM cards up to a total of 8 megabytes. The 2-megabyte memory board sold by The Micro Forge is a typical example of external memory you can add to your Amiga.

THE AMIGA 1300 GENLOCK BOARD

The Amiga 1300 Genlock board, sold by Commodore, lets you mix the graphics and sound from your Amiga with the images and sound coming from an outside source like a television set, a VCR, or a laser disc player. It plugs into the RGB ports and audio ports of the Amiga console, where it receives the video signal coming from the RGB port and the audio signals coming from the audio ports. The Genlock board has its own jacks that accept input from an external video source and an external stereo audio source. It also provides an RGB output for you to plug into your RGB monitor, a composite video output to plug into a VCR for recording the images on the monitor, and left and right audio output jacks to connect to your monitor speakers or to an audio or video recorder.

By using a switch on the Genlock board, you can choose which signal will come from the board's output ports. If you choose the external signal, you'll see and hear whatever video and audio sources you've connected to the external signal jacks. If you choose the Amiga's signal, you see and hear just the graphics and sound generated by the Amiga. If you choose both signals, then you see and hear the external signals and the Amiga's signals mixed together.

When you choose to mix the external signals and the Amiga's signals together, the Genlock board puts the Amiga's images on top of the external images. Wherever there is background color in the Amiga's picture, the external picture shows through. The audio signals are mixed together so you can hear both simultaneously.

Consider an example: If you connect the output of a television receiver with the input of the Genlock board and choose to mix the signals with the Amiga's signals, when you run the Workbench you will see the windows, icons, and gadgets of Workbench all

displayed on a moving background of the 5 o'clock news, or whatever the television receiver is receiving. The TV background replaces the normal blue background you see in Workbench. You would also hear the audio portion of the 5 o'clock news on your monitor speakers. If the Amiga beeps at you as you work, you'll hear the beep mixed in with the news.

The Genlock board is very useful for titling videotapes, among other things. If you connect one VCR to Genlock's input and a second to the Genlock's composite-video output, you can play a videotape on the first VCR, add overlaid graphics on the Amiga, and record the two mixed together on the second VCR. When you consider that the effects created using Deluxe Video or Aegis Animator will work with Genlock, all you have to do is lay down background color in your Amiga videos wherever you want the videotaped images to show through. You can then have titles scrolling over your own family videos.

LASER DISC PLAYERS

A laser disc player is one useful source of an external video signal to use with a Genlock board. The laser disc player uses a laser to read video signals embedded in the laser disc much like a compact disc player uses a laser to read audio signals on a compact disc. You can't record images on a laser disc player like you can on a VCR, but using one does have several advantages: It puts out sharp, clear pictures and stereo high-fidelity sound, and it provides a wide variety of special effects such as slow motion, reverse motion, and freeze-frame; and even more interesting, you can control the way it plays with your Amiga.

Pioneer Video makes most of the laser disc players sold in the United States. The Pioneer CLD-900, an advanced model, plays both laser discs and compact discs, so you can use it for audio or video. It has the special-effect capabilities available in most laser disc players: adjustable slow motion, fast forward and reverse (at three times normal playing speed), forward and reverse scan (a high-speed fast forward and reverse), freeze-frame, and individual frame search. The individual frame search is very powerful. Laser discs produced in the CAV format (the format that laser disc players need to use special effects) can store up to 54,000 individual frames, each frame a full video picture. You can ask the laser player to jump to and display any one of those frames.

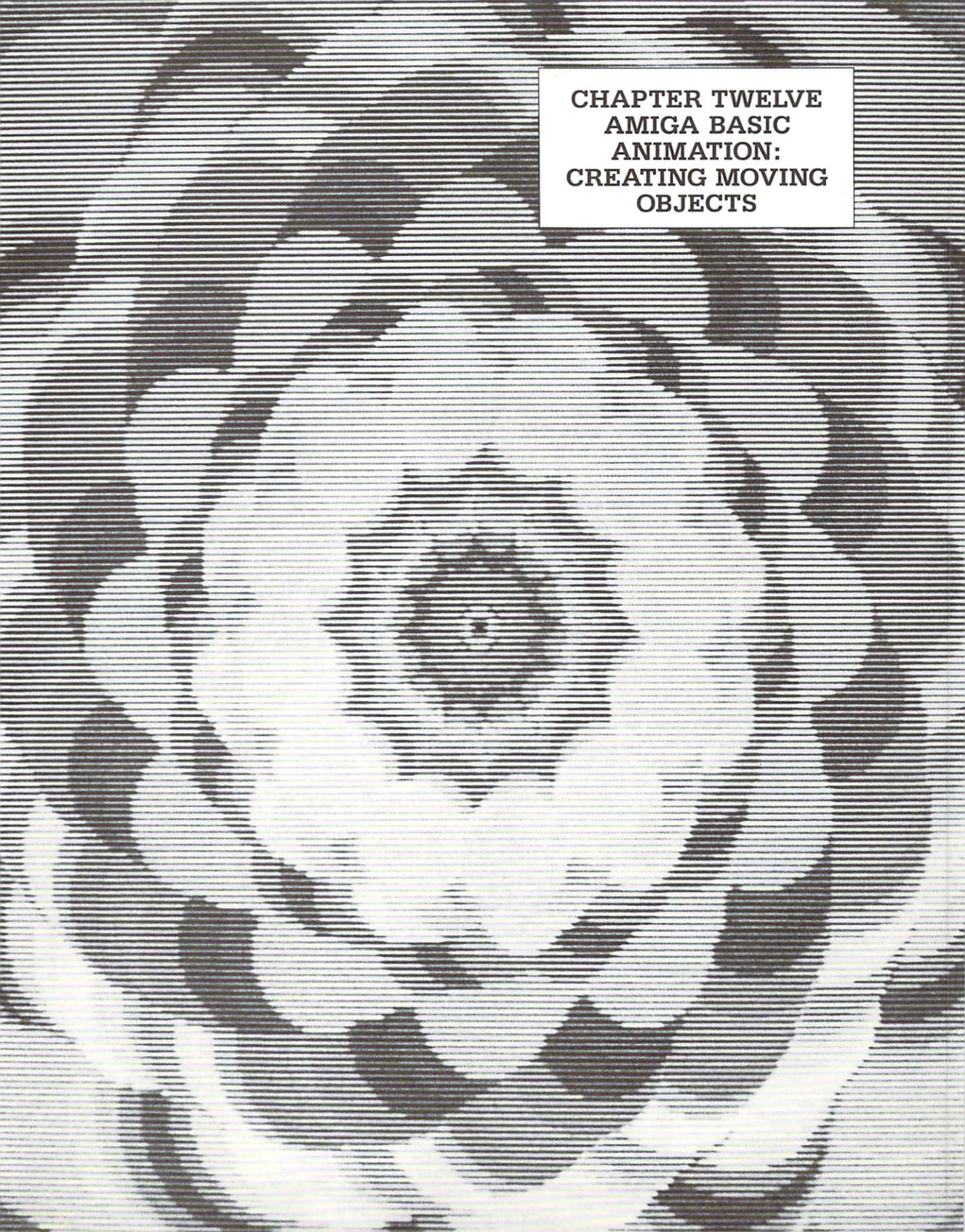
The CLD-900 has an I/O port in the back that you can use to connect a Pioneer IU-04 computer interface. The IU-04 connects to the serial port of the Amiga, and acts as a computerized remote control. The Amiga can send simple commands through its serial port to control the laser disc player in any way you can with the laser disc's own controls.

The Amiga can be used in combination with a laser disc player, the IU-04 interface, and a Genlock board to create some interesting educational programs. For example, you might start with an interesting laser disc like one of the Space Archive laser discs sold by Video Vision Associates. Each Space Archive disc has filmed sequences of moon landings, shuttle launches, and satellite missions, as well as hundreds of still pictures, stored one per frame.

The Amiga can play the laser disc through the Genlock board on the Amiga's monitor. While the disc plays, the Amiga can put up captions on the screen, point out items of interest on the screen, and even make comments in its synthesized voice. At any point in the disc, the Amiga can use the IU-04 to freeze the action. It can ask the viewer if he wants to jump to any other section of the disc, or it might ask questions in a short quiz. It can then move on to another disc section, depending on the user's input.

No existing software application works with a laser disc and Genlock this way, but the hardware and the disc are available, and you can use Amiga BASIC to create a program that can do just what was described.

In this chapter, you've learned some techniques for using Deluxe Video to create your own Amiga videos. You've also taken a tour of some of the other animation hardware and software available for the Amiga. If you're interested in learning more about Amiga animation, read on—the next two chapters describe the many Amiga BASIC animation commands, and tell you quite a bit about how the Amiga creates its animation.



**CHAPTER TWELVE
AMIGA BASIC
ANIMATION:
CREATING MOVING
OBJECTS**

Amiga BASIC contains a wide variety of statements and functions that allow you to move objects around your monitor screen in different directions at varying speeds. It can pass one object over another object, and make objects disappear and reappear at your command. It keeps track of every object on the screen, and checks to see if they have collided with each other or with the edges of the window they're moving in. Amiga BASIC provides you with the tools you need to create interesting animation.

The first step in creating animation is to draw the objects you want to animate with the Object Editor, a BASIC program included on your Amiga BASIC disk. The Object Editor can save the objects to disk for later use in your BASIC programs. When you write a BASIC program, you can use BASIC animation statements to bring the objects back into memory from disk, put objects in the output window, set the speeds of the objects, start the objects moving, stop the objects, and remove them from the screen. You can also include other statements that read the speeds and locations of objects on the screen, and keep track of their collisions.

In this chapter you'll use the Object Editor program to create objects, and you'll also try out some of the BASIC animation statements that put objects on the screen and move them around. In Chapter 13, you'll learn how to control the objects' motion, how to make one object pass over another, and how to control collisions to create convincing animation.

USING THE OBJECT EDITOR

Your first task in creating animation with BASIC is to draw objects with the Object Editor. If you look at the contents of the Amiga BASIC disk, you should see a drawer labeled `BasicDemos`. In that drawer are a variety of BASIC programs that demonstrate the powers of Amiga BASIC. One of those programs is labeled `ObjEdit`: This is the Object Editor.

To use the Object Editor, start it just as you would any other BASIC program: Point to the icon and double-click, or choose `Open` from the `Project` menu from within BASIC. Chapter 7 of the Amiga BASIC manual, "Creating Animated Images," gives directions on how to use the Object Editor. You should read that chapter to find out how to select colors, how to use the drawing tools, and how to save and recall objects from disk.

MODIFYING THE OBJECT EDITOR

The Object Editor on your Amiga BASIC disk was written to work on a 256K Amiga, and is limited to creating objects just two bit planes deep (a four-color limit) in a screen that uses mode 2 resolution. Since Amiga BASIC can use objects of all different depths and resolutions, you should modify the Object Editor program if you have a 512K Amiga so the Object Editor will create objects of all different depths and resolutions. Fortunately, the program is easy to change—all you have to do is make some minor modifications.

First, open the Object Editor. If it's running, stop it and then open the List window. Widen the List window so you can see the full length of the program lines. If you scroll through the program, you'll see that it's quite long, but divided up into short sections, most of them starting with a label and ending with a blank line.

As you modify the Object Editor, you'll replace a few subsections and alter others. To view a particular subsection, just select the Output window and type **list**, followed by a space and the name of the program section. The List window will list the program starting at the first line of that program section. You can then alter the subsection or replace it entirely.

To modify the Object Editor, follow these instructions exactly:

1. Count the lines in the program listing from the beginning of the program. Replace lines 30 to 38 (they start with *scrn* = -1 and end with *WINDOW 1,,(0,0) - (WinX,WinY),31,scrn*) with this new subsection:

SetScreen:

```
INPUT "Resolution mode (1-4)?", res
INPUT "Depth (mode 1: 1-5, mode 2 & 3: 1-4, mode 4: 1-3)?", depth
IF depth < 1 THEN depth = 1
IF res = 4 THEN
    wide = 640: high = 400
    IF depth > 3 THEN depth = 3
ELSEIF res = 3 THEN
    wide = 320: high = 400
    IF depth > 4 THEN depth = 4
ELSEIF res = 2 THEN
    wide = 640: high = 200
    IF depth > 4 THEN depth = 4
ELSE
```

(continued)

```

        wide = 320: high = 200
        IF depth > 5 THEN depth = 5
    END IF
    WinX = wide - 9: WinY = high - 15
    scrn = 1
    SCREEN scrn, wide, high, depth, res
    WINDOW 2, "Object Editor", (0,0) - (WinX,WinY), 0, scrn

```

- List the subsection *StartOver*:. At the end of the subsection, insert this line between the lines *MENU RESET* and *CLS*:

```
SCREEN CLOSE scrn
```

- List the subroutine *InitConstant*:. Change the seventh line of the subroutine from *StatusLine = 20* to:

```
StatusLine = 23
```

- List the subroutine *InitFile*:. Change the third line of the subroutine from *IF Depth = 2 THEN* to:

```
IF Depth = 2 AND res = 1 THEN
```

- List the subroutine *PrintColorBar*:. Replace the entire subroutine with this modified version:

```

PrintColorBar:
    COLOR CurrentColor
    LOCATE 19, 1: PRINT "Color:";
    ColorBar = WINDOW(5) - 10
    COLOR 1
    x = 70: newbar = 0: barcount = 0: barend = 640
    FOR i = 0 TO maxColor
        z = newbar + ColorBar
        LINE (x,z) - (x + 20,z + 10), i, bf
        LINE (x,z) - (x + 20,z + 10), i, b
        x = x + 20
        IF x + 20 > WINDOW(2) THEN
            newbar = newbar + 10
            barcount = (x - 70) / 20
            barend = x: x = 70
        END IF
    NEXT i
    RETURN

```

6. List the subroutine *CheckColor*:. Replace the entire subroutine with this modified version:

```
CheckColor :
  IF CurrentY < ColorBar THEN RETURN
  IF CurrentY > ColorBar + newbar + 10 THEN RETURN
  IF CurrentX < 70 THEN RETURN
  IF CurrentX > barend THEN RETURN
  i = INT((CurrentX - 70) / 20)
  i = i + INT((CurrentY - ColorBar) / 10) * barcount
  IF i > maxColor THEN RETURN
  CurrentColor = i
  GOSUB PrintColorBar
  RETURN
```

7. Save this modified Object Editor under a new name like **NewEdit**.
8. Be sure your default text font for BASIC is set to 80 columns. The text in the new Object Editor won't fit on the screen if you use the 60-column font. If it isn't set to 80, quit BASIC, use Preferences in the Workbench to change the font, and then restart BASIC.

DRAWING AN OBJECT

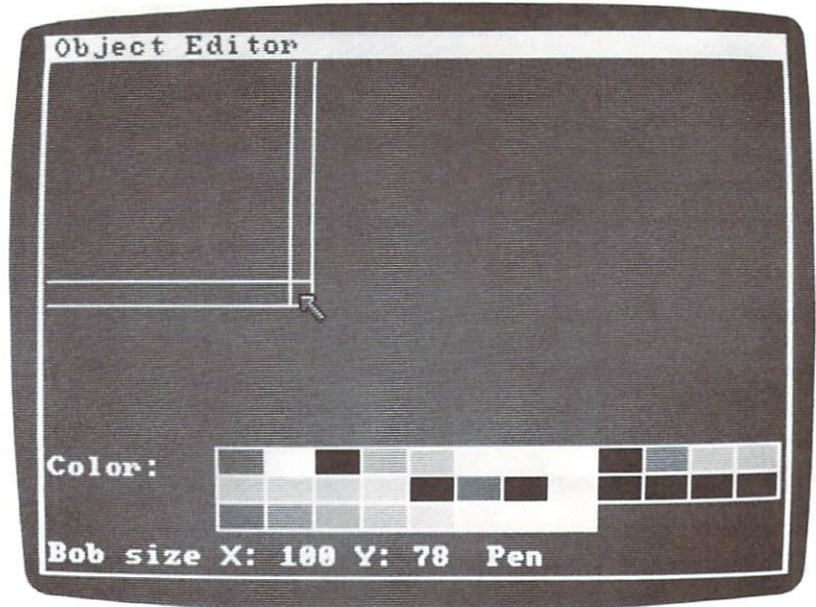
You can use the modified Object Editor to create both sprites and bobs. (See Chapter 10 for more information about sprites and bobs if you don't know what they are.) Run the modified Object Editor program. When it starts, the program will ask you first for a resolution mode and then for a depth. If you're going to draw a sprite, enter mode 1 and depth 2. (You may have to click in the window first to make the window active.)

If you're going to draw a bob, choose a resolution mode that matches the resolution mode you want to use in your animation program. Choose a screen depth that gives you the number of colors you want to work with. (Recall that different bit-plane depths give you a different number of colors to work with: 1 bit plane = 2 colors; 2 bit planes = 4 colors; 3 bit planes = 8 colors; 4 bit planes = 16 colors; and 5 bit planes = 32 colors.)

Once you've chosen a resolution mode and depth, the editor screen appears, unless you chose a resolution of mode 1 and a depth of 2, in which case you're asked to choose between editing a sprite and editing a bob. Once you've made your choice, the editor screen (seen in Figure 12-1 on the next page) will appear and you can begin to draw your object.

Figure 12-1.

The Object Editor's editor screen, here set up for a mode 1 resolution with five bit planes.



Drawing a sprite

When you draw a sprite, the drawing area of the Object Editor has a fixed width that limits the maximum width of the sprite. That's because sprites are limited in width by the hardware that draws them on the screen. The height of the drawing area is adjustable, though, so you can change its size to fit the height of the sprite you want to create.

Once you've set the height, you have a choice of four colors with which to draw the sprite. Keep in mind that the first color in the Object Editor's color bar, the background color, will be transparent when the sprite appears in a window. Any areas of the sprite you colored using the background color (or didn't color at all, since by default all the pixels are the background color) will let the playfield and any other objects beneath the sprite show through.

When you finish drawing your sprite, you can save it to disk using the **Save** command in the **File** menu. It will help jog your memory later if you add **.spr** at the end of the name of each of the sprites you create. For example, you might name a sprite shaped like a pencil **Pencil.spr**. Later, when you look at all the names of the different objects you created and stored on disk, you can identify the sprites by reading the suffixes.

Take some time now to draw a sprite in the shape of a bee and store it on disk. Name it **Bee.spr**. You can use it in the program examples in Chapter 13.

Drawing a bob

You can draw a bob using any of four different resolutions and anywhere from two to 32 different colors, depending on the depth you set. The bob can be a variety of heights and widths. You can change the size of the drawing area by stretching it horizontally and vertically to the size you want. The Object Editor won't let you stretch the drawing area to a size that will create a bob too large for memory.

The colors that appear in the color bar at the bottom of the screen are the default colors contained in the color registers for that screen. When you choose colors to draw your bob, remember that the first color, the background color, is transparent when the bob appears on the screen, just as it is when you use the background color for a sprite.

When you save a bob to disk, you should add a suffix to its name that identifies it as a bob and tells its resolution and how many bit planes it uses. For example, if you create a mode 2 resolution 3-bit-plane bob that is shaped like a car, you might name it **Car.2bob3**. You might also name a mode 1 5-bit-plane bob shaped like a bratwurst sandwich **Bratsan.1bob5**.

Before you finish creating objects with the Object Editor, use it to draw several mode 1 bobs to use in the program examples later in the chapter. Draw a 1-bit-plane bob and save it as **Bird.1bob1**, a 3-bit-plane bob saved as **Flower.1bob3**, and a 5-bit-plane bob saved as **Robot.1bob5**. Don't make them too large! You can use these bobs later. When you've finished using the Object Editor, choose **Quit** to stop running it.

CREATING A PLAYFIELD

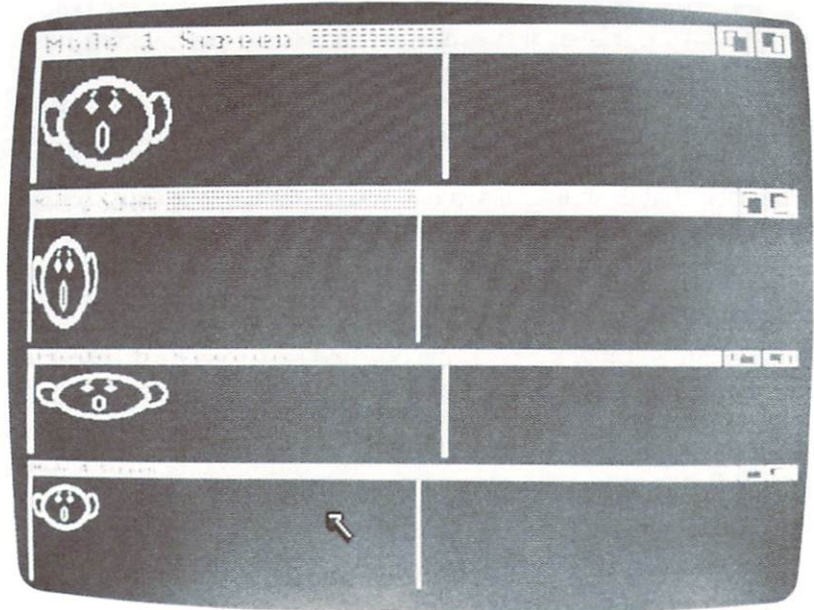
Once you've drawn and saved some bobs and sprites, the first step in creating a BASIC animation program that uses them is to create the playfield, a background on which they can move. To do this, you must first create a screen, and then create an output window using the **SCREEN** and **WINDOW** statements you worked with in Chapter 4. If you want to create a playfield that includes more than just the background color, you can use some of the BASIC drawing and printing statements to create figures or patterns in the output window. Anything you put in the playfield with the drawing and printing statements will be part of the background when you put objects on the playfield with the BASIC animation statements.

SCREEN RESOLUTION

When you set the screen resolution with the SCREEN statement, try to match it to the resolution of the bobs you plan to use. You can put a bob created in one resolution in a screen of a different resolution, but the proportions of the bob will change as a result of the change in proportion of the pixels that build it. For example, a bob created using mode 1 will look half as wide in a mode 2 screen because all the pixels on the screen are half as wide as they are on a mode 1 screen. Figure 12-2 shows a mode 1 bob as it appears in mode 1, mode 2, mode 3, and mode 4 screens.

Figure 12-2.

A mode 1 bob shown from top to bottom in mode 1, mode 2, mode 3, and mode 4 screens.



Sprites look the same no matter what the resolution of the screen they're displayed in. That's because a sprite is not part of the screen itself. It's displayed separately by the Amiga's hardware using mode 1 pixels.

SCREEN DEPTH

To take full advantage of all the colors you selected for a bob, you should create a screen that's as deep or deeper than the deepest bob you want to use. For example, if you want to use two bobs, one two bit planes deep and the other three bit planes deep, you should create a screen at least three bit planes deep to fully accommodate the two bobs.

You can place a bob on a screen with fewer bit planes than the bob if you want, but you won't see all the colors in the bob because the bob is limited to the number of bit planes in the screen. For example, if you put a 5-bit-plane bob with 32 colors on a 2-bit-plane screen that supports just four colors, the bob will be limited to two bit planes, and will have only four colors. Any pixels using colors that are above the range of that screen's depth will be displayed using the background color—only pixels with colors within the depth of the screen will be displayed.

Sprites use their own color registers, so the depth of the screen they appear on won't affect the color of the sprite.

CHOOSING COLORS

When you put bobs and sprites on a screen, the screen uses the colors stored in its color registers to color the bob. These colors can differ from the colors you used to draw the bob, because the Object Editor doesn't save the actual colors you used. Instead, it saves the color-register number of each pixel you used to draw the bob, starting with 0 for the background color, and counting to the right in the Object Editor's color bar. If there is more than one color bar, the count resumes at the left end of the next bar down, and continues to the right. For example, a mode 1 Object Editor screen with 32 colors in the color bar would be numbered as shown in Figure 12-3.

0	1	2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31				

Figure 12-3.

The color registers used in the mode 1, 5-bit-plane Object Editor screen.

To use the colors in the Object Editor that you'll use in your screen, you can further modify the revised Object Editor program: List the subsection of the program named *SetScreen* and insert the same PALETTE statements at the beginning of the subsection that you will use to set the new colors for your screen. (Amiga BASIC's **Cut** and **Paste** commands in the Edit menu work very well for this.) When you run the Object Editor, it will use your new colors.

Sprites use a different color scheme than bobs do. When the Object Editor saves a sprite to disk, it automatically saves the first color in the sprite (the second Workbench color) as turquoise, the

second color (the third Workbench color) as black, and the third color (the fourth Workbench color) as orange. Later, when you recall the sprite back to the screen in an Amiga BASIC program, the sprite appears with those colors.

The Amiga uses color registers 17 through 19 to store the colors of the first two sprites on the screen, color registers 21 through 23 for the next two sprites, color registers 25 to 27 for the next two sprites, and color registers 29 to 31 for the last two sprites. When BASIC loads a sprite from disk, it assigns the colors stored with the sprite to the color registers that sprite will use, changing any colors that might already be there. If you're using a screen with 16 colors or less, this has no effect on the screen colors. If you're using a 32-color screen, the colors on the screen that use the same color registers as the sprite uses will change as soon as a sprite appears.

If you want to create sprites with colors other than turquoise, black, and orange, you can list the section of the Object Editor titled *SaveFile*: and change the 20th, 21st and 22nd lines of the section. These lines assign three hexadecimal RGB values to the sprite disk file, one for each color of the sprite. The value in parentheses following MKI\$ on each line contains a three digit hexadecimal number (preceded by an "&H") that specifies the amount of red, green, and blue that sprite color will contain. The first digit of each number (the one on the left) is the red value, the second digit of each number is the green value, and the third digit of each number is the blue value of that sprite color. Each digit represents the proportion, or strength, of that color on a scale of 0 to 15 (which, in hexadecimal, is 0 to F), so a value of FFF would produce a white color (all three colors at maximum strength), and a value of 000 would produce a black color (all three colors "turned off").

WINDOW SIZE

Any objects and playfield drawings you create have to appear in a window. You can use any type and size of window as your output window, but to make the most of the animation, you should have plenty of room for the objects to move around. A type 0 window—a full-screen window with no title bar, gadgets, or window refresh—gives you the most room to work with. It also keeps the window from being resized or dragged, activities that interfere temporarily with animation in the window.

As an example, the following statements create a playfield for animation statements you'll use later in the chapter. The SCREEN statement creates a mode 1 screen that is five bit planes deep. The WINDOW statement creates a type 0 window without a title bar on the screen to serve as the playfield. The following FOR...NEXT loop creates 50 boxes of random colors and sizes to create an interesting background in the playfield:

```
MakePlayfield:
  SCREEN 1, 320, 200, 5, 1
  WINDOW 2, , , 0, 1
  FOR i = 1 TO 50
    x1 = INT(WINDOW(2) * RND)
    x2 = INT(WINDOW(2) * RND)
    y1 = INT(WINDOW(3) * RND)
    y2 = INT(WINDOW(3) * RND)
    LINE (x1,y1) - (x2,y2), INT(32 * RND), B
  NEXT i
```

This section of program lines, titled *MakePlayfield*, creates a playfield for the mode 1 sprites and bobs you created earlier using the Object Editor. Figure 12-4 shows the playfield. You can use this program section in later program examples.

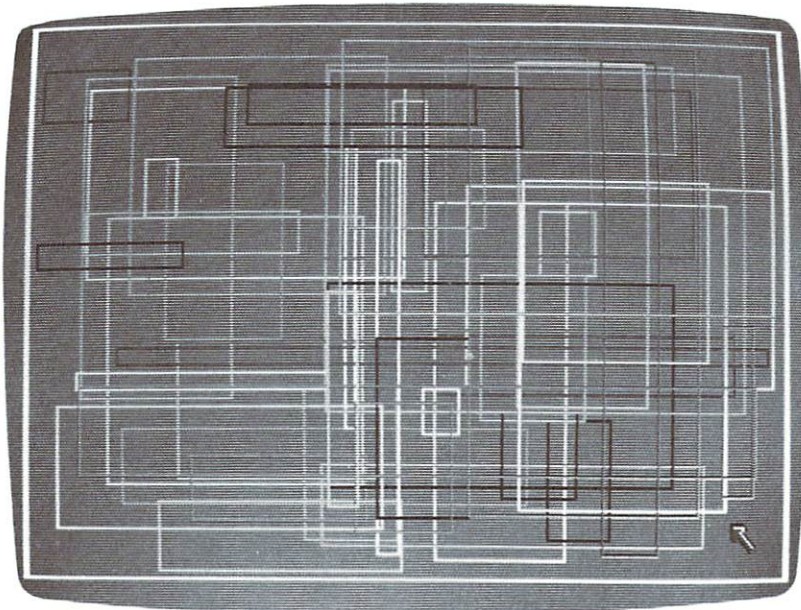


Figure 12-4.

A mode 1 playfield.

PUTTING OBJECTS ON THE PLAYFIELD AND MOVING THEM

Once you've created a screen and opened a window to create a playfield, you can put objects on the playfield and move them with a set of Amiga BASIC animation statements. These BASIC statements work together, and you should use them in the following order to get good results:

1. Bring an object that you saved on disk using the Object Editor into the Amiga's memory with the OBJECT.SHAPE statement.
2. Specify where you want to place the object on the playfield with the OBJECT.X and OBJECT.Y statements.
3. Use the OBJECT.ON statement to make the object appear at the location you specified.
4. Set the speed and direction of an object's motion with the OBJECT.VX and OBJECT.VY statements.
5. Start objects moving with the OBJECT.START statement.
6. Speed up or slow down an object with the OBJECT.AX and OBJECT.AY statements.
7. Stop an object's motion with the OBJECT.STOP statement.
8. Make an object disappear from the playfield with the OBJECT.OFF statement.
9. Erase an object from the Amiga's memory with the OBJECT.CLOSE statement.

This order isn't cast in concrete—for example, you can use an OBJECT.STOP statement before you use OBJECT.AX and OBJECT.AY statements—but for the most part the order is important. For example, if you try to use any of the OBJECT statements without first using the OBJECT.SHAPE statement, you won't get any results. Likewise, it's important to set an object's speed before you use OBJECT.START, and it's also important to start an object moving before you try to stop it with the OBJECT.STOP statement.

As you go through the animation statements in this section, the program examples for each statement often don't do anything by

themselves, since the individual statements need to work with other statements to create and move objects. To make the examples work, you must add them to other program sections to create a complete program, and then run the program. The instructions preceding each program example will tell you how to combine the sections together into a complete program that you can run.

THE OBJECT.SHAPE STATEMENT

The OBJECT.SHAPE statement can perform two different tasks: It can create an object in the Amiga's memory from the information about an object stored on disk by the Object Editor, or it can duplicate an object already in memory. Creating duplicate objects is described in a later section. To put an object into the Amiga's memory, OBJECT.SHAPE uses this format:

```
OBJECT.SHAPE object ID number, object definition string
```

The object ID number can be any integer from 1 up to as many objects as will fit in the Amiga's memory. The object definition string is a string expression that contains the data necessary to create an object.

When BASIC executes OBJECT.SHAPE, it creates an object from the data in the object definition string and assigns it to the specified object ID number. You can then use that object ID number with other OBJECT statements to identify the object. If you use two OBJECT.SHAPE statements that both use the same ID number, the second statement will remove from memory the object that was created with the first statement, then create a new object using the same ID number.

When you save an object to disk using the Object Editor, it saves the object in the format needed by the object definition string for defining the object. This makes it easy to bring an object into your BASIC program—you don't have to worry about defining an object because the Object Editor has already done it for you.

To get the object data stored on disk into the object definition string, you must first open the disk file created by the Object Editor with an OPEN statement, then assign the disk file contents to a string with an INPUT\$ statement, and then close the disk file with a CLOSE statement. The following example opens the disk file for a bob named *Cigar.1bob3*, creates object number 1 using the cigar-bob data, then closes the file:

```
OPEN "Cigar.1bob3" FOR INPUT AS 1
OBJECT.SHAPE 1, INPUT$(LOF(1),1)
CLOSE 1
```

In the first line, the OPEN statement opens a channel of communication numbered "1" between the disk file and the Amiga's memory. In the second line, OBJECT.SHAPE creates an object numbered 1 using the string created by the INPUT\$ statement. The INPUT\$ statement creates the string by reading all the way to the end of the file (found by LOF(1)) that's open on communication channel 1. The third line closes communication channel 1. If you'd like more information about OPEN, CLOSE, INPUT\$, and LOF(), consult the Amiga BASIC manual.

OBJECT DATA STORED IN DRAWERS

When BASIC opens the object's disk file, it expects to find the object file in the same Workbench drawer that the program it is running is stored in. If you stored the object file in another drawer, BASIC won't find it and will return a *File not found* error message. For example, assume you created a Workbench drawer called **Animation**. Inside the Animation drawer, you created two more drawers: **Programs** and **Objects**. You store all your BASIC animation programs in the Programs drawer, and all disk files you create with the Object Editor in the Objects drawer. When you run a program stored in the Programs drawer, the OPEN statement will look for the object file to open in the Programs drawer, and won't find it, since it's in the Objects drawer.

To make the program safe to use no matter where you store it, you can specify drawers in the OPEN statement. In the previous example, you could use the statement

```
OPEN ":Animation/Objects/Cigar.1bob3" FOR INPUT AS 1
```

in place of the previous OPEN statement. The longer filename, which is called a pathname, starts with a colon that asks OPEN to begin searching on the same disk you're using to run BASIC. It then asks OPEN to look inside the Animation drawer, where it should find and look inside the Objects drawer, where it should find "Cigar.1bob3." By using a pathname, you can store the object files in a completely different drawer from the program, run the program, and still have the OPEN statement find the object file that you want.

To create your own object pathname, always start with a colon or the name of the disk drive (such as **df1:**), then follow it with the names of the drawers in which you store your objects, in order from the first drawer you see on the Workbench to the final drawer that contains your objects, ending with the name of your object file. Separate the drawer names and the object filename with slashes.

You should also keep in mind that objects created by the Object Editor will automatically be stored in the drawer containing the Object Editor unless you specify otherwise using a pathname. To avoid confusion, you should always copy your objects into the drawer you intend to use them from immediately after you quit the Object Editor.

To make some objects for your example program, add the following program section to the playfield example you typed in earlier. (These object filenames assume your objects are in the same drawer as your program.) This program section, which is named *MakeObjects*, creates three objects numbered 1, 2, and 3 in the Amiga's memory:

```
MakeObjects:
  OPEN "Flower.1bob3" FOR INPUT AS 1
  OBJECT.SHAPE 1, INPUT$(LOF(1), 1)
  CLOSE 1
  OPEN "Robot.1bob5" FOR INPUT AS 1
  OBJECT.SHAPE 2, INPUT$(LOF(1), 1)
  CLOSE 1
  OPEN "Bee.spr" FOR INPUT AS 1
  OBJECT.SHAPE 3, INPUT$(LOF(1), 1)
  CLOSE 1
```

THE OBJECT.X AND OBJECT.Y STATEMENTS

Once you've used the OBJECT.SHAPE statement to create an object in memory, you can set it in place on the playfield using the OBJECT.X and OBJECT.Y statements. They use this format:

```
OBJECT.X object ID number, x coordinate
```

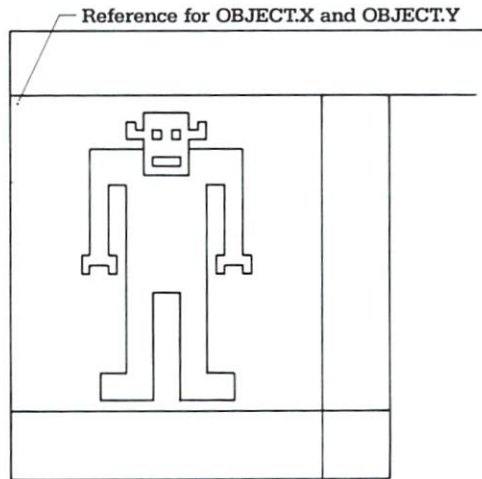
```
OBJECT.Y object ID number, y coordinate
```

The object ID number is any ID number you used earlier in the program to create an object with OBJECT.SHAPE. The x and y coordinates can be any integers from -32768 to +32767.

To understand how these statements position an object in the playfield, you must know how they view the boundaries of an object. When you create an object with the Object Editor, even if it's a small object in the center of the drawing area, you are actually creating an object with rectangular boundaries the size of the Object Editor's drawing area. Chances are the boundaries are invisible; you probably left background space around the object you drew that doesn't show up on the playfield. When OBJECT.X and OBJECT.Y position an object, they use the upper left corner of the object's rectangular boundaries, as you can see in Figure 12-5 (on the next page).

Figure 12-5.

The upper left corner pixel of an object is used by OBJECT.X and OBJECT.Y to position the object in the playfield.



With this in mind, you should try to create objects in the Object Editor as close to the upper left corner of the drawing area as possible. This makes positioning the object easier and more accurate, especially if the object is much smaller than the drawing area. Don't worry about the invisible boundary interfering with the way objects collide—Amiga BASIC uses the boundary only to position the object. It ignores the boundary when objects collide, and pays attention only to the non-background pixels in the object.

To position an object using OBJECT.X and OBJECT.Y, find the address of the pixel on the playfield where you want to position the upper left corner of the object's boundary. Then follow OBJECT.X with the ID number of the object and the x (horizontal) coordinate of the pixel address, then follow OBJECT.Y with the ID number of the object and the y (vertical) coordinate of the pixel address. For example, to position the upper left corner of object 1 on pixel (34,50), use these two statements:

```
OBJECT.X 1, 34  
OBJECT.Y 1, 50
```

If you don't specify x and y coordinates for an object with OBJECT.X and OBJECT.Y statements, BASIC positions the object at pixel (0,0).

It's also possible to position an object entirely outside the boundaries of the window you're using as a playfield. If you use x and y coordinates beyond the range of the window, the object will be positioned beyond the window boundaries. For example,

```
OBJECT.X 1, 1028  
OBJECT.Y 1, 50
```

would put the object far off the right side of the window, even if it was a full-width mode 4 window with 640 pixels across.

To position the three objects you made earlier for the example program, add this program section at the end of the program:

```
PlaceObjects:  
  OBJECT.X 1, 0: OBJECT.Y 1, 0  
  OBJECT.X 2, 150: OBJECT.Y 2, 0  
  OBJECT.X 3, 0: OBJECT.Y 3, 150
```

These program lines position object 1 in the upper left corner of the playfield, object 2 in the upper middle of the playfield, and object 3 in the lower left corner of the playfield. You won't be able to see them, however, until the objects are "turned on" with the OBJECTON statement.

THE OBJECTON STATEMENT

Positioning an object with OBJECTX and OBJECTY won't make the object visible on the playfield. To do that, use the OBJECTON statement to make objects visible wherever they have been positioned on the playfield. It uses this format:

```
OBJECTON object ID number, object ID number, ...
```

The object ID numbers correspond to the object ID numbers of objects created earlier in the program with OBJECTSHAPE. You can follow OBJECTON with as many object ID numbers as will fit in one program line, as long as you separate them with commas. You can also use just one object ID number, or use OBJECTON without any ID numbers at all, in which case all the objects created with OBJECTSHAPE statements earlier in the program will become visible on the screen in the positions you set with OBJECTX and OBJECTY. If you didn't specify an object's position, it will appear at pixel (0,0) in the upper left corner of the playfield.

If you follow `OBJECT.ON` with one or more object ID numbers, only the objects you specify will become visible on the playfield. Unspecified objects will remain invisible. For example, the statement

```
OBJECT.ON 3, 4
```

will make objects 3 and 4 appear on the playfield.

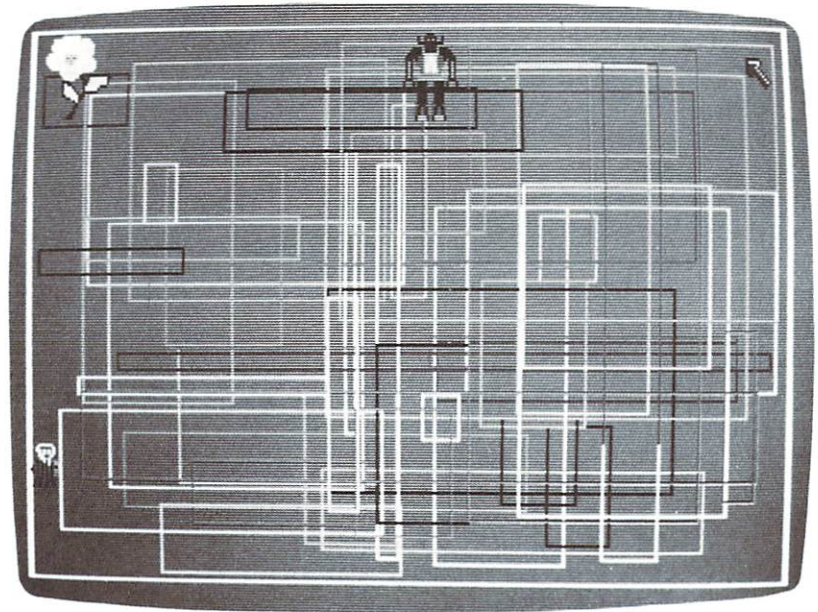
To make all the objects in your example program appear on the playfield, add this line to the end of the *PlaceObjects*: program section:

```
OBJECT.ON
```

To see the objects, try running the program. You should see the objects appear where you positioned them, as shown below in Figure 12-6.

Figure 12-6.

Three objects positioned on a playfield with `OBJECT.X`, `OBJECT.Y`, AND `OBJECT.ON` statements.



THE `OBJECT.VX` AND `OBJECT.VY` STATEMENTS

To set an object on the playfield in motion, use the `OBJECT.VX` and `OBJECT.VY` statements to set the horizontal and vertical velocity of the object. They use these formats:

```
OBJECT.VX object ID number, x velocity
```

```
OBJECT.VY object ID number, y velocity
```

The object ID number is any of the ID numbers used by earlier OBJECT.SHAPE statements in the program to create objects. The x and y velocities can be any integers from -32768 to +32767.

The x and y velocities measure the speed of the object in pixels per second. The x velocity measures the horizontal speed, and the y velocity measures the vertical speed. A positive x velocity value moves the object right, and a negative x velocity value moves the object left. A positive y velocity value moves the object down, and a negative y velocity value moves the object up. For example,

```
OBJECT.VX 1, 23
OBJECT.VY 1, 42
```

sets a velocity that moves object number 1 23 pixels to the right and 42 pixels down per second.

```
OBJECT.VX 1, -52
OBJECT.VY 1, -4
```

moves object number 1 52 pixels to the left and 4 pixels up per second.

If you want to be precise, you can apply some basic rules of trigonometry to calculate the direction that the object will move, given the x and y velocity values for the object. For example, consider the object mentioned before with an x velocity of 23 and a y velocity of 42. By dividing the x value by the y value, then calculating the arctangent of the result, you can determine that the object will travel at a 28.70 degree angle to the right of a straight-down direction. Figure 12-7 illustrates this.

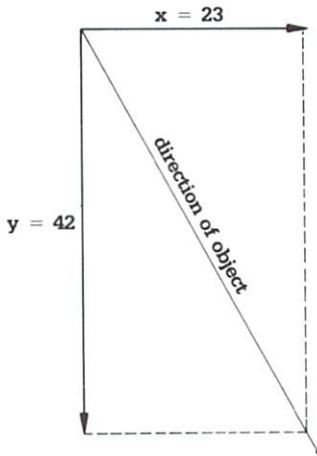


Figure 12-7.

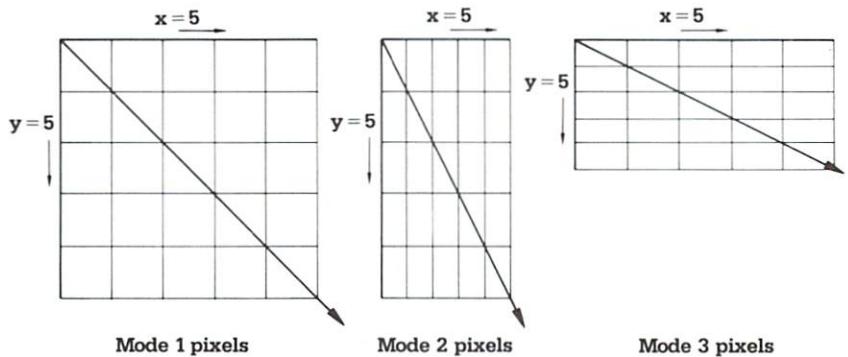
The direction of an object traveling 23 pixels to the right and 42 pixels down per second.

Direction of object

Keep in mind as you set an object's direction with x and y velocity values that the proportions of the individual pixels in the screen are an important factor in figuring the final direction of the object. For example, an x velocity of 5 and an equal y velocity of 5 in a mode 1 screen will move the object at a 45-degree diagonal. The same velocities in a mode 2 screen will move the object at approximately a 30-degree diagonal: The pixels are half as wide as they are high, so 5 pixels down per second is twice the distance of 5 pixels to the right. On a mode 3 screen, x and y velocities of 5 will move the object at approximately a 60-degree angle: The pixels are half as high as they are wide, so 5 pixels to the right per second is twice the distance of 5 pixels down.

Figure 12-8.

The x and y velocity values for a mode 1 screen will produce different results on mode 2 and mode 3 screens.



On a mode 4 screen, x and y velocities of 5 will move the object in a 45-degree diagonal as they would in a mode 1 screen, but the velocity would be halved compared to a mode 1 screen because the pixels are only half as wide and half as high as they are in a mode 1 screen.

Although you can in theory set an object's speed to 32767 pixels per second, large numbers like these aren't practical. If you're using a 320-by 200-pixel playfield, any speed much over 1000 pixels per second moves the object so fast you can't even see it move.

To set the speeds of objects in your example program, add this program section to the end of the program:

```
MoveObjects:  
  OBJECT.VX 1, 10: OBJECT.VY 1, 10  
  OBJECT.VX 2, 0: OBJECT.VY 2, 10  
  OBJECT.VX 3, 60: OBJECT.VY 3, 0
```


These three program lines move object 1 diagonally down and to the right at a 45-degree angle, object 2 straight down, and object 3 quickly from left to right across the playfield.

THE OBJECT.START STATEMENT

Setting the velocity of an object doesn't start the object moving. You use the OBJECT.START statement to set the object in motion. It uses this format:

```
OBJECT.START object ID number, object ID number, . . .
```

The object ID number is any of the ID numbers of objects created earlier in the program with the OBJECT.SHAPE statement. You can follow OBJECT.START with as many ID numbers as will fit on a single program line, all separated by commas, or you can use OBJECT.START with a single ID number, or without ID numbers at all, in which case every object on the playfield is set in motion using the velocities set with OBJECT.VX and OBJECT.VY statements. If you specify object ID numbers, only the objects specified will start moving. If you don't assign an x or y velocity value to an object, it won't move when you use the OBJECT.START statement because objects have default x and y velocities of 0.

When you put an object in motion with OBJECT.START, it moves until it collides with another object or with one of the borders of the playfield window, or until it is stopped with an OBJECT.STOP statement (discussed shortly). Once stopped, an object remains at rest until another OBJECT.START statement starts it again. All objects stop when the program ends, so unless you set up an endless loop at the end of the program or otherwise prevent the program from ending, you'll see all the moving objects suddenly stop in their tracks as soon as BASIC has finished executing all the program lines.

To set all the objects in your example program in motion, add this line to the end of the *MoveObjects*: program section:

```
OBJECT.START
```

If you want to see how the program works now, add this one-line program section at the end of the program and run it:

```
Loop: GOTO Loop
```

This line creates an endless loop so your objects will keep moving until they collide with something.

THE OBJECT.AX AND OBJECT.AY STATEMENTS

After setting an object's velocity and moving it with OBJECT.START, it moves in one direction at a steady speed. To change an object's direction and speed, you can use the OBJECT.AX and OBJECT.AY statements. They use these formats:

```
OBJECT.AX object ID number, x acceleration
```

```
OBJECT.AY object ID number, y acceleration
```

The object ID number is any of the ID numbers used by earlier OBJECT.SHAPE statements in the program to create objects. The x and y acceleration can be any integer from -32768 to $+32767$.

The x and y accelerations measure the change in velocity of the object in pixels per second. The x acceleration value affects horizontal acceleration, the y acceleration value affects vertical acceleration. A positive x acceleration value accelerates the object to the right, a negative x acceleration value accelerates the object to the left. A positive y acceleration value accelerates the object down, and a negative y acceleration value accelerates the object up.

To see how these two acceleration statements affect the velocity of an object, consider an object (ID number 2) that has an x velocity of 25 pixels per second, and a y velocity of 25 pixels per second. It moves on a 45-degree diagonal right and down toward the lower right corner of the playfield. Use these two statements:

```
OBJECT.AX 2, -2  
OBJECT.AY 2, -5
```

to slowly change the direction of the object up and to the left.

The following chart shows how the velocity of object 2 in the preceding example would change over a period of 10 seconds:

	x velocity	y velocity
Start speed:	25	25
1 second:	23	20
2 seconds:	21	15
3 seconds:	19	10
4 seconds:	17	5
5 seconds:	15	0
6 seconds:	13	-5
7 seconds:	11	-10
8 seconds:	9	-15
9 seconds:	7	-20
10 seconds:	5	-25

After 10 seconds, the object has curved to change its direction up and a little to the right.

By applying acceleration changes to the straight directions you set with OBJECT.VX and OBJECT.VY, you can move your object in some very interesting curves. Experiment to see what you can do!

To add some interesting acceleration to your example program, run the program after adding this short program section just before the *Loop*: section at the end of the program:

```
ChangeMotion:  
  OBJECT.AY 3, -2
```

You should see object 3 curve up toward the top of the screen.

THE OBJECT.STOP STATEMENT

To stop a moving object, you use the OBJECT.STOP statement. It uses this format:

```
OBJECT.STOP object ID number, object ID number, . . .
```

You can specify any number of objects by listing their ID numbers, separated by commas, after OBJECT.STOP. OBJECT.STOP will stop only the objects you specify. If you use OBJECT.STOP without specifying an object ID number, it will stop all the objects that are in motion.

Insert this program section into your example program just before the *Loop*: section and run the program to see object 2 stop in the middle of the playfield:

```
StopObject:  
  FOR i = 1 TO 3000: NEXT i  
  OBJECT.STOP 2
```

The FOR...NEXT loop makes the program wait so object 2 can move partially down the screen before it's stopped by the OBJECT.STOP statement.

THE OBJECT.OFF STATEMENT

To make an object invisible while it's located on the playfield, use the OBJECT.OFF statement. It uses this format:

```
OBJECT.OFF object ID number, object ID number, . . .
```

Like OBJECT.STOP, you can specify any number of objects by listing their ID numbers, in which case OBJECT.OFF makes invisible only the objects you specify, or you can use OBJECT.OFF without specifying any objects, in which case it makes all the objects in the program disappear.

OBJECT.OFF automatically stops the motion of each object it makes disappear. An object can't move while it's invisible, unless you assign it a new location with the OBJECT.X and OBJECT.Y statements. If you don't change the object's location before you use OBJECT.ON to make the object reappear, the object will reappear and resume moving from exactly the same location where it disappeared.

To see how OBJECT.OFF works, replace the *StopObject:* section of your example program with the following section, and then run the program:

```
BlinkObjects:
  FOR i = 1 TO 3000: NEXT i
  OBJECT.OFF
  FOR i = 1 TO 3000: NEXT i
  OBJECT.ON
```

The first FOR...NEXT loop lets the objects move normally for a while. When the loop is finished, you should see all the objects in motion disappear briefly while the second FOR...NEXT loop executes, and then reappear in the same spot they disappeared, resuming their previous motion.

THE OBJECT.CLOSE STATEMENT

Each time you create an object in memory with the OBJECT.SHAPE statement, BASIC reserves several kilobytes of memory to store the object. To regain that memory for other uses, you can clear an object from memory when it's no longer in use by using the OBJECT.CLOSE statement. It uses this format:

```
OBJECT.CLOSE object ID number, object ID number, ...
```

You can specify individual objects by their ID numbers, in which case OBJECT.CLOSE clears just the objects you specify from memory, or you can use OBJECT.CLOSE without specifying any objects, in which case it clears all objects from memory.

If you close an object that's still moving on the playfield, it will stop at once and disappear. Any subsequent OBJECT statements that refer to that object ID number won't work until you create a new object with the same ID number using OBJECT.SHAPE. If you want to bring the object back into memory, you'll have to use the OBJECT.SHAPE statement to recreate it again.

Objects remain in memory even after an animation program has stopped running, so it's a good idea to clear all the objects at the end of the program. Since many animation programs use an

endless loop to keep the animation moving, the user has to use the **Stop** command in the **Run** menu to stop the program, so there's no possibility of closing the objects within the program. After stopping the program, you can use `OBJECT.CLOSE` as an immediate command in the Output window, but this doesn't provide the user with an efficient means of clearing memory. It's even better to create an alternative way for the user to quit the program, so the program can finish up by closing screens and windows and clearing objects from memory.

CREATING DUPLICATE OBJECTS

If you write animation programs that use many objects, you can save memory and work by duplicating existing objects. You can use the `OBJECT.SHAPE` statement to duplicate any object already in memory, and then use the `OBJECTPLANES` statement to change the colors of either the original or the duplicate objects.

USING OBJECT.SHAPE TO DUPLICATE OBJECTS

When you use `OBJECT.SHAPE` to create a new object in memory, you assign the object's shape and color to an object ID number with an object definition string that contains the object's data. If you replace the object definition string with the object ID number of an object that's already in memory, `OBJECT.SHAPE` will create a new object that uses the same shape and colors as the original object you specified. `OBJECT.SHAPE` uses this format to duplicate an object:

```
OBJECT.SHAPE duplicate object ID number , original object ID number
```

The duplicate object ID number can be any integer from 1 to as many objects as memory will hold. The original object ID number must be the ID number of an object that has already been created with the `OBJECT.SHAPE` statement.

When you create a duplicate object, it takes much less memory to store the object than it takes to store an original object created with the Object Editor. That's because both the original object and its duplicate object use the same memory to store shape and color. Each duplicate object you create requires only enough additional memory to store information such as velocity and position.

Each duplicate object is still a separate entity when you move it around the playfield. It has its own ID number, and you can assign a unique position, velocity, and acceleration to each duplicate object. You can turn each duplicate object on and off, start and stop it, and close it without affecting the other duplicate objects or

the original object. However, if you close the original object, the duplicate objects will freeze in their tracks and refuse to respond to any statement that asks them to move or change location, so avoid closing an original object before you close its duplicates.

The following program creates a playfield and uses the OBJECT.SHAPE statement to make three duplicates of an original object. It places the four objects in the four corners of the playfield, and moves them toward the center of the playfield:

```
PlayField:
    SCREEN 1, 320, 200, 5, 1
    WINDOW 2, , , 0, 1

MakeOriginal:
    OPEN "Flower.1bob3" FOR INPUT AS 1
    OBJECT.SHAPE 1, INPUT$(LOF(1), 1)
    CLOSE 1

MakeDuplicates:
    FOR i = 2 TO 4
        OBJECT.SHAPE i, 1
    NEXT i

PlaceObjects:
    OBJECT.X 1, 0: OBJECT.Y 1, 0
    OBJECT.X 2, 0: OBJECT.Y 2, 150
    OBJECT.X 3, 250: OBJECT.Y 3, 150
    OBJECT.X 4, 250: OBJECT.Y 4, 0
    OBJECT.ON

MoveObjects:
    OBJECT.VX 1, 20: OBJECT.VY 1, 10
    OBJECT.VX 2, 20: OBJECT.VY 2, -10
    OBJECT.VX 3, -20: OBJECT.VY 3, -10
    OBJECT.VX 4, -20: OBJECT.VY 4, 10
    OBJECT.START

Loop: GOTO Loop
```

USING OBJECT.PLANES TO CHANGE COLORS

Duplicate objects are the same color and shape as the original object. To visibly distinguish the original object from its duplicate, you can change the colors of the original or duplicate object to a new set of colors by using the OBJECT.PLANES statement. OBJECT.PLANES works only on bobs; it won't work on sprites. It uses this format:

```
OBJECT.PLANES object ID number, PlanePick value,  
              PlaneOnOff value
```


The object ID number can specify any object already created with the `OBJECT.SHAPE` statement, whether it's an original or a duplicate object. The `PlanePick` and `PlaneOnOff` values can be any integers from 0 to 31 in a 5-bit-plane screen, 0 to 15 in a 4-bit-plane screen, 0 to 7 in a 3-bit-plane screen, and 0 to 3 in a 2-bit-plane screen. The `OBJECT.PLANES` statement has no practical effect on a 1-bit-plane screen. You can omit either the `PlanePick` value or the `PlaneOnOff` value, in which case they're set to 0, but you can't omit both. If you omit the `PlanePick` value, be sure to hold its place with a comma.

`OBJECT.PLANES` makes direct use of the graphics-animation libraries in the system software by setting two bit-masks named `PlanePick` and `PlaneOnOff` using the values you specify. To understand exactly how these masks work, you need to know binary arithmetic and precise details of how the Amiga stores its graphics data. Rather than tackle topics that go beyond the range of this book, a general description of how the `OBJECT.PLANES` statement works and some rules of thumb may help you to use the statement effectively without having to learn advanced topics. If you want to get more information, you can find detailed accounts of `PlanePick` and `PlaneOnOff` in the *Amiga ROM Kernal Manual*, available through your Amiga dealer.

To get a general understanding of `OBJECT.PLANES`, consider how the Amiga stores colors in a playfield. The color-register number of each pixel is a binary number that's stored in memory with one bit in each of the playfield's bit planes. For example, in a 5-bit-plane playfield, each pixel has a 5-bit number, stored with one bit in each bit plane. In a 3-bit-plane playfield, each pixel has a 3-bit number, stored one bit per bit plane.

Now consider what happens when the Amiga moves a 2-bit-plane bob through a 5-bit-plane playfield. To move the bob around the playfield, the Amiga erases the pixels where the bob will be located, and fills them with the color-register numbers of the bob's pixels. When the bob moves away, the Amiga restores the original pixel numbers in the playfield. Since the bob pixels are only two bit planes deep, their color-register numbers use only the bottom two bit planes of the playfield pixels they occupy. Since the top three bit planes aren't used, the Amiga fills them with zeros.

When you set a `PlanePick` value using `OBJECT.PLANES`, you tell the Amiga to spread the bob's bit planes throughout the playfield bit planes. For example, with a 2-bit-plane bob in a 5-bit-plane playfield, you can specify that the bob pixels' color-register numbers will be stored in playfield bit planes 2 and 4 instead of bit planes 1 and 2 (the two bottom bit planes). The Amiga will fill in

the remaining planes (1, 3, and 5), with zeros. This results in a new set of color-register numbers for each pixel in the bob and changes the color of the bob.

You can further change the bob's colors by setting the PlaneOnOff mask. PlaneOnOff fills the bit planes that are unused by the bob's pixels with either ones or zeros, depending on the PlaneOnOff value you used. This lets you insert ones or zeros in the bob pixels' unused bit planes, and lets you further change the bob colors to a new set of colors.

One way to get usable results from OBJECTPLANES without calculating original bob colors and resulting bob colors through binary bit-plane conversions is to use the trial-and-error method: Write a short program that puts an original and duplicate bob on the playfield and use OBJECTPLANES with different values for PlanePick and PlaneOnOff to change the color of either the original or the duplicate bob. Run the program, and look at the results. You can change the PlanePick and PlaneOnOff values until you get the colors you want. Of course, if you're working with a 5-bit-plane screen, there are over a thousand different combinations of values, most of which won't give you the results you want. For example, many of the PlanePick values you can use with OBJECT.PLANE will actually decrease the number of colors you see in a bob. There are several ways to limit the number of possible values to a set of values that works well:

- To avoid losing colors, don't use OBJECT.PLANE on a bob if the bob has as many bit planes as the screen in which it appears. It will only reduce the number of colors in the bob, or at best leave the colors alone. For example, OBJECT.PLANE won't work well for a 4-bit-plane bob in a 4-bit-plane screen.
- The PlaneOnOff value sets a mask that works with whatever value is in the PlanePick mask. Many PlaneOnOff values have no effect at all when they work with certain PlanePick values. It's always best to try a PlanePick value by itself to see how the bob's color changes, and then to add the PlaneOnOff value to see how it changes the PlanePick colors.
- To set a bob back to its original colors, set the PlanePick and PlaneOnOff values at 0.

Setting PlanePick and PlaneOnOff values

What follows are tables of PlanePick and PlaneOnOff values for bobs of different bit-plane depths. These tables are designed to reduce your trial and error time, as the values they contain have been calculated to produce meaningful results. The PlanePick values in the table won't decrease the number of colors in the bob, and the table's PlaneOnOff values will change the colors set by the PlanePick value.

1-bit-plane bobs

	PlanePick values	PlaneOnOff values that produce results
...in a 2-bit-plane screen	1	0, 2
	2	0, 1
...in a 3-bit-plane screen	1	0, 2, 4, 6
	2	0, 1, 4, 5
	4	0, 1, 2, 3
...in a 4-bit-plane screen	1	0, 2, 4, 6, 8, 10, 12, 14
	2	0, 1, 4, 5, 8, 10, 12, 14
	4	0, 1, 2, 3, 8, 9, 10, 11
	8	0, 1, 2, 3, 4, 5, 6, 7
...in a 5-bit-plane screen	1	0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30
	2	0, 1, 4, 5, 8, 9, 12, 13, 16, 17, 20, 21, 24, 25, 28, 29
	4	0, 1, 2, 3, 8, 9, 10, 11, 16, 17, 18, 19, 24, 25, 26, 27
	8	0, 1, 2, 3, 4, 5, 6, 7, 16, 17, 18, 19, 20, 21, 22, 23
	16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

2-bit-plane bobs

	PlanePick values	PlaneOnOff values that produce results
...in a 3-bit-plane screen	3	0, 4
	5	0, 2
	6	0, 1
...in a 4-bit-plane screen	3	0, 4, 8, 12
	5	0, 2, 8, 10
	6	0, 1, 8, 9
	9	0, 2, 4, 6
	10	0, 1, 4, 5
	12	0, 1, 2, 3
...in a 5-bit-plane screen	3	0, 4, 8, 12, 16, 20, 24, 28
	5	0, 2, 8, 10, 16, 18, 24, 26
	6	0, 1, 8, 9, 16, 17, 24, 25
	9	0, 2, 4, 6, 18, 20, 22, 24
	10	0, 1, 4, 5, 16, 17, 20, 21
	12	0, 1, 2, 3, 16, 17, 18, 19
	17	0, 2, 4, 6, 8, 10, 12, 14
	18	0, 1, 4, 5, 8, 9, 12, 13
	20	0, 1, 2, 3, 8, 9, 10, 11
24	0, 1, 2, 3, 4, 5, 6, 7	

3-bit-plane bobs

	PlanePick values	PlaneOnOff values that produce results
...in a 4-bit-plane screen	7	0, 8
	11	0, 4
	13	0, 2
	14	0, 1
...in a 5-bit-plane screen	7	0, 8, 16, 24
	11	0, 4, 16, 20
	13	0, 2, 16, 18
	14	0, 1, 16, 17
	19	0, 4, 8, 12
	21	0, 2, 8, 10
	22	0, 1, 8, 9
	25	0, 2, 4, 6
	26	0, 1, 4, 5
28	0, 1, 2, 3	

4-bit-plane bobs

	PlanePick values	PlaneOnOff values that produce results
...in a	15	0, 16
5-bit-plane	23	0, 8
screen	27	0, 4
	29	0, 2
	30	0, 1

An OBJECT.PLANES example

The best way to use these different values is to find the table of values for the bob and screen you're working with. Plug the different values into the OBJECT.PLANES statement and run the program to see what colors you come up with. Trial and error will eventually find the colors you want. For example, the following program section uses three of the sets of values that change the color of three 3-bit-plane bobs in a 5-bit-plane screen:

```
ColorBobs:
  OBJECT.PLANES 2, 19, 8
  OBJECT.PLANES 3, 19, 0
  OBJECT.PLANES 4, 14, 1
```

If you insert this section just before the *PlaceObjects:* section in the last example program, when you run the program you'll see that each of the three duplicate bobs is colored differently.

You've now learned how to draw your own sprites and bobs with the Object Editor. With the Amiga BASIC statements you've learned in this chapter, you can put those bobs in the Amiga's memory, and then put them on the screen, moving them in different direction at different speeds. You can also duplicate and recolor bobs. In the next chapter, you'll be able to make the bobs move where and how you want them to, and be able to control how they react when they hit other bobs.



**CHAPTER THIRTEEN
AMIGA BASIC
ANIMATION:
CONTROLLING
MOTION**

In the last chapter, you put objects on a playfield, instructed them to move, and watched them move like drifting boats until you stopped them or until they ran into another object or the playfield border. Amiga BASIC can control moving objects with much more finesse than that. By constantly checking the position and speed of each object, BASIC can tell when the object reaches a desired (or undesired) location or speed, and can then take appropriate actions. By checking to see if any of the objects collide with another object or the playfield border, it can act on the collision to let one object pass over another, make the object change direction, remove it from the playfield, or perform any number of other actions.

In this chapter, you'll learn how to use the Amiga BASIC animation statements that check and control motion and collisions, and you'll see how to put the statements together to make objects move with rhyme and reason. You'll also find out how to control which object passes over another object to create 2½-D animation, and how to avoid collisions between objects. The end of the chapter has information about interesting miscellaneous animation abilities of Amiga BASIC: how to scroll a section of the playfield, and how to use the PUT and GET statements to create true sequenced-image animation.

CREATING A MOTION BOUNDARY WITH THE OBJECT.CLIP STATEMENT

One of the simplest ways to control the motion of moving objects in a playfield is to create a motion boundary. When you first create an output window to use as the playfield, BASIC automatically sets the motion boundaries as the borders of the window itself. You can use the OBJECT.CLIP statement to reset the motion boundaries to enclose a smaller area of the playfield within the output window so the moving objects won't move out of the area and interfere with text or graphics displayed in another part of the playfield. You can

also use OBJECT.CLIP to create motion boundaries that are beyond the borders of the output window so objects can move beyond the window borders, out of sight, before they hit a boundary.

THE OBJECT.CLIP STATEMENT

The OBJECT.CLIP statement uses this format:

```
OBJECT.CLIP(corner address) - (corner address)
```

The corner addresses are like the standard pixel addresses you use with drawing statements, except that the x and y coordinates can be any integer from -32768 to +32767, making addresses located outside of the output window like (-1045,78) possible. Like the two addresses used to create a box with the LINE statement, these two corner addresses set the location of two opposing corners of a rectangle.

Any objects moving within the OBJECT.CLIP rectangle will move until they collide with one of its sides or another object. Any objects located outside the rectangle won't move even if they're given an OBJECT.START command.

In the following program example, the OBJECT.CLIP statement creates an invisible rectangle in the middle of the screen. When you run the program, an object is positioned within the rectangle and instructed to move until it is stopped by one of the boundaries. Figure 13-1 (on the next page) shows you how it works.

MakePlayfield:

```
SCREEN 1, 320, 200, 5, 1  
WINDOW 2, , , 0, 1
```

MakeObject:

```
OPEN "Bird.1bob1" FOR INPUT AS 1  
OBJECT.SHAPE 1, INPUT$(LOF(1), 1)  
CLOSE 1
```

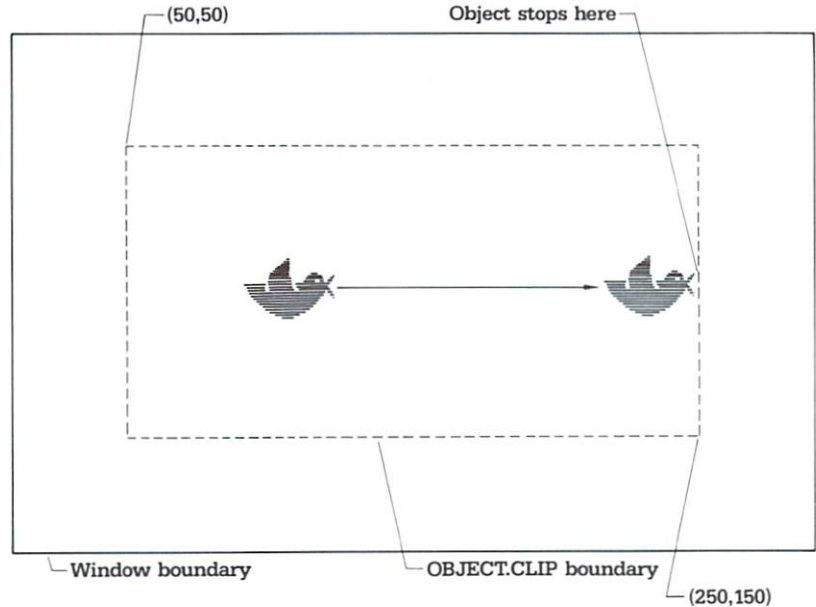
MoveObject:

```
OBJECT.X 1, 100: OBJECT.Y 1, 100  
OBJECT.VX 1, 25  
OBJECT.CLIP(50,50) - (250,150)  
OBJECT.ON  
OBJECT.START
```

Loop: GOTO Loop

Figure 13-1.

An object is stopped by boundaries set by the OBJECT.CLIP statement.



READING AN OBJECT'S LOCATION AND VELOCITY

To get more control over moving objects than you can get with OBJECT.CLIP, it's important for BASIC to be able to read the location and velocity of each moving object at any moment. Amiga BASIC has four functions that return the x and y coordinates and the x and y velocities of any object on the playfield. You can write a BASIC program to test the values the functions return, and then act accordingly.

THE OBJECT.X() AND OBJECT.Y() FUNCTIONS

OBJECT.X() and OBJECT.Y() are two functions that return the x and y coordinates of an object's location in a playfield. They use these formats:

```
OBJECT.X(object ID number)
```

```
OBJECT.Y(object ID number)
```

Don't confuse these two functions with the OBJECT.X and OBJECT.Y statements you read about in the last chapter. The functions here use parentheses around the object ID number and don't include coordinate values—this is what distinguishes them as functions, not statements. The object ID number is the ID number of any object created previously in the program by an OBJECT.SHAPE statement.

OBJECT.X() and OBJECT.Y() return the x and y coordinates of the pixel address of the upper left corner of the object's rectangular boundary. (See the description of the OBJECT.X and OBJECT.Y statements in Chapter 12 for more information about how the upper left corner of an object's boundary is determined.) OBJECT.X() returns the x (horizontal) coordinate, OBJECT.Y() returns the y (vertical) coordinate.

Since OBJECT.X() and OBJECT.Y() are functions that don't actually perform any task when used alone in a program line, they must be used as a value assigned to a variable or used within a statement in a program line. For example, the following two program lines assign the x and y coordinates of object 3 to the variables x3 and y3:

```
x3 = OBJECT.X(3)
y3 = OBJECT.Y(3)
```

If the upper left corner of object 3's boundary was located at pixel (45,102) in the playfield when BASIC executed these two statements, x3 would then be equal to 45, and y3 would be equal to 102.

THE OBJECT.VX() AND OBJECT.VY() FUNCTIONS

To read the velocity and direction of motion of any moving object in a program, you can use the OBJECT.VX() and OBJECT.VY() functions with these formats:

```
OBJECT.VX(object ID number)
OBJECT.VY(object ID number)
```

Again, don't confuse these two functions with the OBJECT.VX and OBJECT.VY statements you read about in the last chapter. The OBJECT.VX() and OBJECT.VY() functions use parentheses around the object ID number, and don't include a velocity value. The *object ID number* is the ID number of an object created earlier in the program by an OBJECT.SHAPE statement. OBJECT.VX() returns the horizontal (x) velocity in pixels per second of the object you specify; OBJECT.VY() returns the vertical (y) velocity in pixels per second.

Like OBJECT.X() and OBJECT.Y(), OBJECT.VX() and OBJECT.VY() are functions, so they can't be used alone in a program line. They must be assigned to a variable or used as a value with another statement. For example, the following two program lines use the OBJECT.VX() and OBJECT.VY() functions to read the x and y velocities of object 2, and use the OBJECT.VX and OBJECT.VY statements to assign them as the velocities for object number 3:

```
OBJECT.VX 3, OBJECT.VX(2)
OBJECT.VY 3, OBJECT.VY(2)
```

AN EXAMPLE USING LOCATION AND VELOCITY FUNCTIONS

If you create a series of statements within a program loop that constantly reads the location and velocity of objects, you can use an IF...THEN statement within that loop to branch the program to another action if the statements within the loop read a desired location or velocity of an object. In the following program example, two objects are moving on the playfield. Object 1 moves in a diagonal from the upper left corner toward the lower right corner. Object 3 moves from left to right across the lower part of the screen, starting slowly and accelerating as it moves.

The section of the program labeled *Test:* is a loop that uses OBJECT.Y() to see how far object 1 has moved down the screen, and OBJECT.VX() to see how fast object 3 is moving across the screen. If object 1 moves to a position with a y coordinate higher than 99, the program jumps to a subroutine that stops the object. If object 3 accelerates to an x velocity faster than 50 pixels per second, then the program jumps to a subroutine that stops object 3 from accelerating any further. Figure 13-2 shows how this works.

```
MakePlayfield:
    SCREEN 1, 320, 200, 5, 1
    WINDOW 2, , , 0, 1

MakeObjects:
    OPEN "Bee.spr" FOR INPUT AS 1
    OBJECT.SHAPE 1, INPUT$(LOF(1), 1)
    CLOSE 1
    OPEN "Flower.1bob3" FOR INPUT AS 1
    OBJECT.SHAPE 3, INPUT$(LOF(1), 1)
    CLOSE 1

PlaceObjects:
    OBJECT.X 1, 0: OBJECT.Y 1, 0
    OBJECT.X 3, 0: OBJECT.Y 3, 150
    OBJECT.ON
```

(continued)


```

MoveObjects:
  OBJECT.VX 1, 10: OBJECT.VY 1, 10
  OBJECT.VX 3, 5: OBJECT.AX 3, 1
  OBJECT.START

Test:
  IF OBJECT.Y(1) > 99 THEN GOSUB Stop1
  IF OBJECT.VX(3) > 60 THEN GOSUB Steady3
  GOTO Test

Stop1:
  OBJECT.STOP 1
  RETURN

Steady3:
  OBJECT.AX 3, 0
  RETURN

```

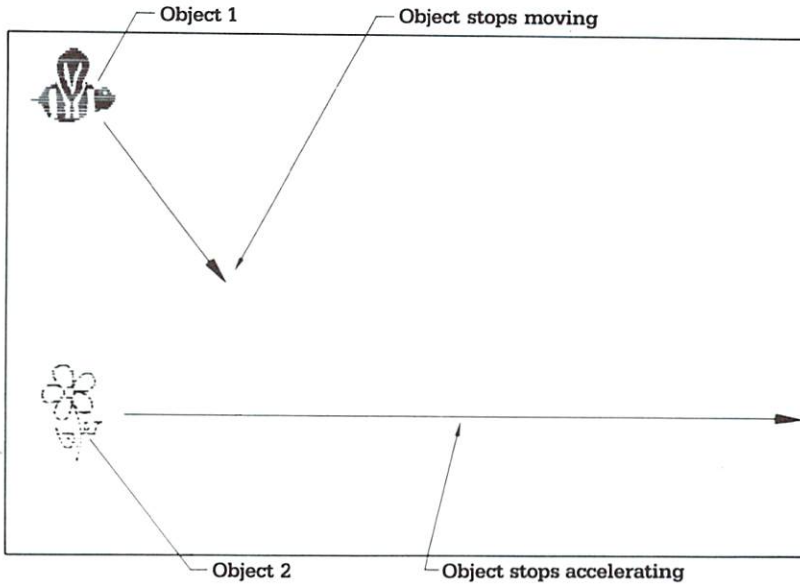


Figure 13-2.

You can use BASIC's location and velocity functions to check an object's speed and position, then have BASIC react accordingly.

HANDLING OBJECT COLLISIONS

Controlling object collisions is very important in Amiga BASIC animation. As you saw in the last chapter, whenever an object collides with another object or a boundary, it automatically stops its motion unless you include statements that tell BASIC to perform

some other action in the event of a collision. With the OBJECT.HIT statement, you can ask BASIC to ignore a collision, in which case the objects will just pass each other. You use the OBJECT.PRIORITY statement to tell BASIC which object should appear on top as they pass. If you do allow objects to collide, you can use the COLLISION ON and ON COLLISION GOSUB statements with the COLLISION() function to control how the objects react when they collide.

THE OBJECT.HIT STATEMENT

The OBJECT.HIT statement lets you define objects as different types, and lets you decide which types of objects will collide and which types won't collide. It uses this format:

```
OBJECT.HIT object ID number, type value, hit value
```

The object ID number is the ID number of any object created earlier in the program with the OBJECT.SHAPE statement. Both the type value and the hit value can be any integer from -32768 to +32767.

You use the type value to define the specified object as none, any, or all of 16 different types of objects. BASIC interprets type 1 as a window or OBJECT.CLIP boundary. The other 15 types of objects are abstract types—BASIC only knows them by numbers as types 2 through 16—so you can think of them as whatever you wish. For example, in a video-game program you might decide to use type 2 objects as spaceships, type 3 objects as laser beams, and type 4 objects as planets. The following chart lists a value for each of the 16 different object types:

Type 1: 1	Type 5: 16	Type 9: 256	Type 13: 4096
Type 2: 2	Type 6: 32	Type 10: 512	Type 14: 8192
Type 3: 4	Type 7: 64	Type 11: 1024	Type 15: 16384
Type 4: 8	Type 8: 128	Type 12: 2048	Type 16: -32768

To define an object as any of these types except type 1 (defining an object as a boundary won't work—BASIC just ignores it), just choose the types you want, then add their values together to come up with the type value. For example, if you want an object to be defined as both a type 7 and a type 12 object, you add the values for types 7 and 12 together (64 and 2048) to get a result of 2112. That is the type value for that object.

If you want an object to be defined as just one object type, use the value for that one type as the type value. For example, to define an object as just a type 16 object, you use a type value of -32768 . If you use a type value of 0, you tell BASIC that the specified object isn't any of the 16 types. If you want to define an object as all 16 object types, you use the type value of -1 , which is the sum of all 16 values in the table.

The hit value is a number that identifies all the types of objects that an object can collide with. When an object touches a second object in the playfield, BASIC checks the first object's type value to see what type (or types) of object it is. It then checks the second object's hit value to see what types of objects it's supposed to collide with. If the first object is on the second object's hit list, BASIC registers a collision between the two objects. If the first object isn't on the hit list, the two objects pass by without collision.

The hit value uses the same table of values that the type value uses. To set the hit value, you choose the type or types of objects you want to put on the hit list, find their values, then add them together to get the hit value. If you don't want any types of objects or the boundaries on the hit list, then use a hit value of 0. If you want all types of objects on the hit list, then use a hit value of -1 . If you want only boundaries on the hit list, then use a hit value of 1.

When two objects touch on the playfield, it's important for BASIC to determine which is the first object and which is the second object, since BASIC must match the hit value of the first object against the type value of the second object. The rule of collision is that the object that is the upper leftmost of the two objects is the first object. If you want to define an object that won't collide with any other object on the playfield, you have to assign it a type value of 0 and a hit value of 0 or 1, since you can never be sure if it will be the first object that BASIC checks for type, or the second object that BASIC checks for a hit list.

When an object hits a boundary, BASIC always checks the hit list of the object. The boundary has no hit list itself. It has only a type value of 1.

As an example, think of a simple animation with three types of moving objects: bees, flowers waving in the wind, and birds. Bees shouldn't collide with other bees and birds, but should collide with flowers so they can stop for nectar. Birds shouldn't collide with any of the three types of objects, and flowers shouldn't collide with other flowers or birds, but should collide with bees.

If you decide that bees are object type 2, birds are object type 3, and flowers are object type 4, then you can assign type and hit values for the objects. Bees should have a type value of 2 and a hit

value of 9, so they can hit both flowers (8) and boundaries (1). Birds should have a type value of 4, and a hit value of 1, so they can collide with boundaries, but not anything else. Flowers should have a type value of 8 and a hit value of 3, so they can collide with bees (2) and boundaries (1).

The following simple example program puts one bee, one bird, and one flower in motion on the playfield, using those type and hit values. This example uses the sprites and bobs that you drew in Chapter 12. When you run it, you should see the bee (object 1) move diagonally across the screen, while the bird (object 2) and the flower (object 3) move horizontally across the screen. The bee will pass over the bird and stop when it collides with the flower. Figure 13-3 illustrates the objects' actions.

```
MakePlayfield:
    SCREEN 1, 320, 200, 5, 1
    WINDOW 2, , , 0, 1

MakeObjects:
    OPEN "Bee.spr" FOR INPUT AS 1
    OBJECT.SHAPE 1, INPUT$(LOF(1)), 1
    CLOSE 1
    OPEN "Bird.1bob1" FOR INPUT AS 1
    OBJECT.SHAPE 2, INPUT$(LOF(1)), 1
    CLOSE 1
    OPEN "Flower.1bob3" FOR INPUT AS 1
    OBJECT.SHAPE 3, INPUT$(LOF(1)), 1
    CLOSE 1

PlaceObjects:
    OBJECT.X 1, 0: OBJECT.Y 1, 0
    OBJECT.X 2, 0: OBJECT.Y 2, 75
    OBJECT.X 3, 0: OBJECT.Y 3, 150
    OBJECT.ON

IdentifyObjects:
    OBJECT.HIT 1, 2, 9
    OBJECT.HIT 2, 4, 1
    OBJECT.HIT 3, 8, 3

MoveObjectsNew:
    OBJECT.VX 1, 20: OBJECT.VY 1, 20
    OBJECT.VX 2, 20
    OBJECT.VX 3, 20
    OBJECT.START

Loop: GOTO Loop
```

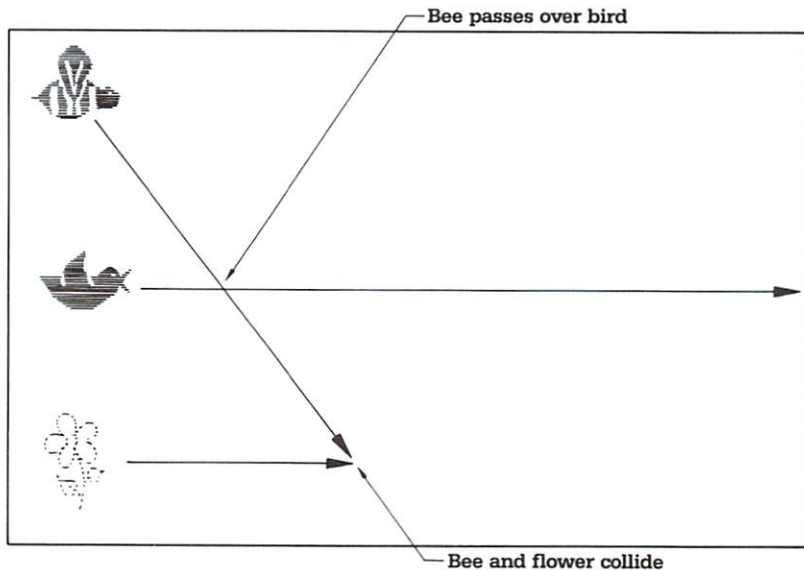


Figure 13-3.

Using OBJECT.HIT to define different types of objects enables a bee to pass over a bird and collide with a flower.

THE OBJECT.PRIORITY STATEMENT

When two objects pass and touch, BASIC has to decide which object will pass over the top of the other. To do so, it uses the same rule it uses to determine which is the first of two colliding objects: By default, the upper leftmost object will pass over the lower rightmost object. If you want to change the passing priority, you can use the OBJECT.PRIORITY statement. It uses this format:

```
OBJECT.PRIORITY object ID number, priority value
```

The object ID number is the ID number of an object created earlier in the program with an OBJECT.SHAPE statement. The object must be a bob; OBJECT.PRIORITY won't work with sprites. The priority value is any integer from -32768 to $+32767$.

When you use OBJECT.PRIORITY to assign a priority value to a specified bob, BASIC remembers that value when two bobs pass each other. The bob with the higher priority value will pass over the top of the bob with the lower priority value. For example, if you assign a priority value of 2 to one bob and 5 to a second bob, the second bob will always pass over the first bob. If two bobs have the same priority, BASIC falls back on the "upper leftmost" rule to decide which bob appears on top.

Consider the previous program example. If you substitute the robot bob for the bee sprite in the *MakeObjects:* program section,

then add this program section just before the *MoveObjectsNew*: section of the last example, when you run the program the robot will pass under the bird instead of over it, because the robot's priority is lower than the bird's:

```
SetPriority:  
    OBJECT.PRIORITY 1, 0  
    OBJECT.PRIORITY 2, 10
```

THE COLLISION ON STATEMENT

Although its name implies that COLLISION ON will enable collisions to occur on the playfield, that's not what it does. Instead, it turns on the collision queue. The collision queue is a section of memory that keeps a record of the collisions that occur on the playfield, in the order they occur.

COLLISION ON also enables other statements and functions to work, like ON COLLISION GOSUB and COLLISION(), a statement and a function that use the collision queue. COLLISION ON uses a very simple format:

```
COLLISION ON
```

Once BASIC executes COLLISION ON, the collision queue stays on until BASIC executes a COLLISION OFF or COLLISION STOP statement (both discussed later in this chapter).

THE ON COLLISION GOSUB STATEMENT

BASIC automatically stops an object when it collides with another object or a border (unless, of course, you've instructed BASIC not to do so). If you want something else to happen when a collision occurs, you can write a subroutine that performs the object action you want, and then use the ON COLLISION GOSUB statement to jump to that subroutine as soon as a collision occurs. ON COLLISION GOSUB uses this format:

```
ON COLLISION GOSUB line name/number
```

The line name/number is the subroutine name or the line number where a subroutine starts.

When BASIC executes ON COLLISION GOSUB in a program, it then moves on to execute the following program statements just as if nothing happened—with one exception. After it executes each

statement, it looks at the collision queue to see if any collisions have occurred. If there are no collisions, BASIC keeps executing more program statements just as it would normally. If a collision occurs, BASIC immediately jumps to the beginning of the subroutine you specified in the ON COLLISION GOSUB statement, executes it, and then returns to execute the statement that follows the last statement it executed before it detected the collision.

If you want to turn off the ON COLLISION GOSUB statement so that BASIC won't jump to a subroutine for every collision that occurs, use:

```
ON COLLISION GOSUB 0
```

The line number 0 turns off the collision checking.

What happens in the subroutine that ON COLLISION GOSUB points to is entirely up to you. Chances are that you will use the COLLISION() function in the subroutine to find out more about the collision before you act on it.

THE COLLISION() FUNCTION

The COLLISION() function returns information from the collision queue. It uses this format:

```
COLLISION( ID number )
```

The ID number can be the object ID number of any object created earlier in the program by an OBJECT.SHAPE statement. It can also be a -1 or a 0.

To best understand how the COLLISION() function works, it's important to know how the collision queue stores collisions. The queue can store up to 16 collisions at one time. If any collisions occur after the queue is filled up, they aren't stored in the queue and leave no record of their occurrence.

The collision queue stores three pieces of information for each collision that it records:

1. The object ID number of the first (upper leftmost) object of two colliding objects. If it's a collision between an object and a boundary, it stores the object ID number of the object.
2. The object ID number of the second (lower rightmost) object in a two-object collision. If it's a collision between an object and a boundary, it stores one of four negative numbers that

indicates which of the four boundaries has been hit. Those four boundary ID numbers are:

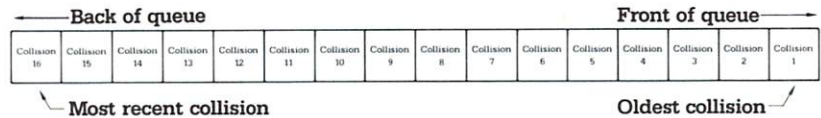
BOUNDARY	ID NUMBER
Top:	- 1
Left:	- 2
Bottom:	- 3
Right:	- 4

- The window number of the window where the collision occurred.

The collision queue stores collisions in the order they occur; that is, the first collision in the queue is the collision that occurred before all the other collisions; the last collision in the queue is the most recent collision. Figure 13-4 shows how this works.

Figure 13-4.

The collision queue.



If you use a 0 as the ID number for COLLISION(), COLLISION() checks the first (oldest) collision in the queue and returns the object ID number of the first object involved in the collision. If you use a -1 as the ID number, COLLISION() returns the window ID number of the window where the collision identified by COLLISION(0) occurred. When COLLISION() checks the queue using -1 or 0 as the ID number, it reads information from the queue without changing the contents of the queue.

If you use an object ID number as the ID number (a positive integer), COLLISION() searches through the collisions in the queue from the front to the back to see if that object was involved in any collision as either the first or second object. If it finds a collision involving that object, COLLISION() returns the object ID number of the other object or the boundary ID number of the boundary that the object collided with. It also removes that collision from the queue, so that all the collisions behind it move forward and leave

another space open for a collision at the end of the queue. If COLLISION() finds no collision in the queue involving the specified object, it returns a 0 and doesn't alter the contents of the queue.

To give you an idea of how COLLISION() works, consider this example. Three successive collisions occur on a playfield while the collision queue is turned on: Object 2 collides with object 1, object 4 collides with the left boundary, and object 3 collides with object 1. If you use

```
PRINT COLLISION(0)
```

to check the contents of the queue, it returns a 2, the object ID of the first object involved in the first collision. If you then use

```
PRINT COLLISION(2)
```

it returns a 1, the object ID number of the object that object 2 hit. It also takes that collision off the collision queue. If you use

```
PRINT COLLISION(1)
```

COLLISION() searches the queue for a collision involving object 1, and finds one at the end of the queue. It returns a 3, the object ID number of the object that object 1 hit, and then removes the collision from the queue. The only collision left in the queue, now at the front of the queue, is the collision between object 4 and the left boundary. If you use

```
COLLISION(4)
```

it will return a -2, the boundary ID number, and removes that collision from the collision queue.

AN EXAMPLE USING COLLISION ON, ON COLLISION GOSUB, AND COLLISION()

The following program example puts three objects on the screen and moves them in different directions. The program section labeled *DetectCollision*: uses COLLISION ON to turn on the collision queue, and ON COLLISION GOSUB to start the program checking for collisions. It names *CheckCollision*: as the name of the line to jump to when a collision occurs.

The subroutine *CheckCollision*: checks to see what kind of collision occurs, and changes the motion of the objects so they

move away from what they hit. The second line of the subroutine gets the object number of the first object in the first collision and stores it in the variable *objectid*. The third line checks it to see if there actually has been a collision—if there hasn't, it returns program execution to the *Loop*: section, which is an endless loop that loops until a collision occurs. The fourth line of *CheckCollision*: reads the ID number of the object that was hit in the first collision in the queue, and assigns it to the variable *hitobject*.

The next two lines of the subroutine check to see if the object has hit the upper or lower boundary (-1 or -3), and if it has, reverses the vertical motion of the object. Lines 7 and 8 check to see if the object has hit the left or right boundary (-2 or -4), and reverses the horizontal motion of the object if it has. Lines 9, 10, and 11 assume that if the object hasn't hit any boundaries, it must have hit another object. They reverse the vertical and horizontal motion of both objects involved in the collision.

The OBJECT.START statement toward the end of the subroutine starts the colliding object or objects in motion again, since BASIC automatically stops them when they collide. The last line of the subroutine returns program execution to the *Loop*: section of the program.

MakePlayfield:

```
SCREEN 1, 320, 200, 5, 1
WINDOW 2, , , 0, 1
```

MakeObjects:

```
OPEN "Bee.spr" FOR INPUT AS 1
OBJECT.SHAPE 1, INPUT$(LOF(1)), 1
CLOSE 1
OPEN "Bird.1bob1" FOR INPUT AS 1
OBJECT.SHAPE 2, INPUT$(LOF(1)), 1
CLOSE 1
OPEN "Flower.1bob3" FOR INPUT AS 1
OBJECT.SHAPE 3, INPUT$(LOF(1)), 1
CLOSE 1
```

PlaceObjects:

```
OBJECT.X 1, 0: OBJECT.Y 1, 0
OBJECT.X 2, 0: OBJECT.Y 2, 150
OBJECT.X 3, 250: OBJECT.Y 3, 0
OBJECT.ON
```

(continued)

```

MoveObjects:
    OBJECT.VX 1, 15: OBJECT.VY 1, 25
    OBJECT.VX 2, 40: OBJECT.VY 2, -30
    OBJECT.VX 3, -15: OBJECT.VY 3, 20
    OBJECT.START

DetectCollision:
    COLLISION ON
    ON COLLISION GOSUB CheckCollision

Loop: GOTO Loop

CheckCollision:
    objectid = COLLISION(0)
    IF objectid = 0 THEN RETURN
    hitobject = COLLISION(objectid)
    IF hitobject = -1 OR hitobject = -3 THEN
        OBJECT.VY objectid, -(OBJECT.VY(objectid))
    ELSEIF hitobject = -2 OR hitobject = -4 THEN
        OBJECT.VX objectid, -(OBJECT.VX(objectid))
    ELSE
        OBJECT.VY objectid, -(OBJECT.VY(objectid))
        OBJECT.VX objectid, -(OBJECT.VX(objectid))
        OBJECT.VY hitobject, -(OBJECT.VY(hitobject))
        OBJECT.VX hitobject, -(OBJECT.VX(hitobject))
    END IF
    OBJECT.START
    RETURN

```

THE COLLISION STOP STATEMENT

There are times in a program when you may want to keep ON COLLISION GOSUB from jumping down to a subroutine on every collision. You can temporarily suspend ON COLLISION GOSUB by using the COLLISION STOP statement. It uses this format:

```
COLLISION STOP
```

When BASIC executes COLLISION STOP in a program after an ON COLLISION GOSUB statement, it keeps the collision queue up to date by adding collisions until the queue is full. It doesn't let ON COLLISION GOSUB know that a collision has occurred so the program execution won't jump to the collision subroutine.

To make ON COLLISION GOSUB work again, you use another COLLISION ON statement. If any collisions have entered the collision queue since the COLLISION STOP statement, they will trigger ON COLLISION GOSUB, and the program execution will immediately jump to the collision subroutine.

THE COLLISION OFF STATEMENT

The COLLISION OFF statement turns off the collision queue entirely. It uses this format:

```
COLLISION OFF
```

When BASIC executes COLLISION OFF, any collisions that take place afterward aren't stored at all, and are forgotten by BASIC. Any collisions that are stored in the queue before COLLISION OFF will remain there. To start storing collisions in the collision queue again, use a COLLISION ON statement.

MORE AMIGA ANIMATION TRICKS

Most of the Amiga BASIC animation statements are OBJECT statements that move sprites and bobs around the playfield. There are two other tricks available through Amiga BASIC to put motion on your screen. One moves the playfield itself, the other creates motion using true sequential-image animation.

SCROLLING THE PLAYFIELD WITH THE SCROLL STATEMENT

You can use the SCROLL statement to define a rectangular area of the playfield and then scroll its contents vertically, horizontally, or diagonally. It uses this format:

```
SCROLL (corner address) - (corner address), x shift, y shift
```

The corner addresses are the same as the corner addresses you use with the OBJECT:CLIP statement. The x and y coordinates can be any integer from -32768 to $+32767$, making addresses like $(-1045,78)$ possible. These two addresses set the location of two opposing corners of a scrolling rectangle. The x shift and y shift are each integers from -32768 to $+32767$. You can use just an x-shift, or just a y-shift, but if you omit the x-shift value, be sure to hold its place with a comma.

The two corner addresses set the opposing corners of a rectangular area on the output window. You can use them to create a scrolling rectangle within the output window that's smaller than the window, or to create a scrolling rectangle that covers the full window. Although you can set addresses far outside the borders of the output window you're using, any addresses outside the window will be treated as if they're on the border of the window. For example, if you create a scrolling rectangle with the corner addresses $(-256, -45) - (600,450)$ in a mode 1 window, you won't actually create a scrolling rectangle larger than the window, you'll create a rectangle the size of the full window.

The x-shift value specifies in pixels how far you want to horizontally move the contents of the scroll rectangle. A positive number means you want to move the contents to the right, a negative number specifies a shift to the left. The y-shift value specifies in pixels how far you want to vertically move the contents of the scroll rectangle. A positive number specifies a shift down, a negative number specifies a shift up. By using the two values together in a SCROLL statement, you can create a diagonal shift. For example, this statement shifts the contents of a scroll rectangle in the upper left corner of the output window, moving them 20 pixels to the right and 30 pixels down:

```
SCROLL (0,0) - (160,100), 20, 30
```

When BASIC executes this statement, the contents shift diagonally toward the lower right corner of the window.

When SCROLL shifts the contents of the scroll rectangle outside the boundaries of the rectangle, it erases them. If you scroll the same rectangle back in the opposite direction, the parts of the rectangle that passed the boundaries return as just background color. For example,

```
SCROLL (0,0) - (160,100), 20, 30  
SCROLL (0,0) - (160,100), -20, -30
```

scrolls the contents of the rectangle past the rectangle boundaries, then returns them to their original position. Figure 13-5 (below) shows the results.

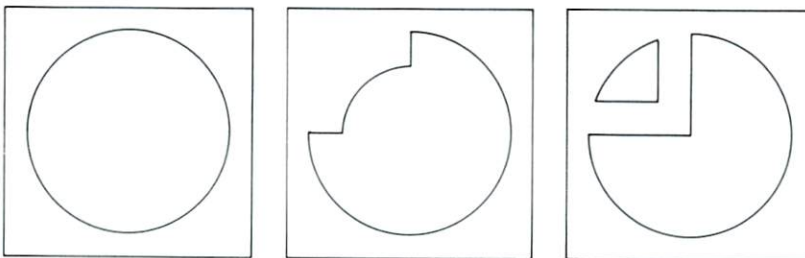


Figure 13-5.

Three different stages of a scroll: The left picture shows the window before scrolling, the middle picture shows the window after the first SCROLL statement, and the right picture shows the window after the last SCROLL statement.

A single SCROLL statement does not actually scroll the contents of the scroll rectangle. It shifts them in one instantaneous motion. To achieve a smooth scrolling effect, you need to use several SCROLL statements in a row with small x-shift and y-shift values.

For example, the following program creates a background, then scrolls a section of the background diagonally back and forth using FOR...NEXT loops:

```
MakePlayfield:
    SCREEN 1, 320, 200, 5, 1
    WINDOW 2, , , 0, 1

FillText:
    WIDTH 40
    FOR i = 1 TO 100
        PRINT "Kumquat ";
    NEXT i

ScrollBlock:
    FOR i = 1 TO 15
        FOR j = 1 TO 20: NEXT j
        SCROLL (100, 25) - (200,110), 1, 1
    NEXT i
    FOR i = 1 TO 15
        FOR j = 1 TO 20: NEXT j
        SCROLL (100, 25) - (200,110), -1,-1
    NEXT i
    GOTO ScrollBlock
```

The first FOR...NEXT loop in the program section *ScrollBlock*: scrolls the contents of the scroll rectangle down and to the right. The little FOR...NEXT loop nested inside it kills some time so the scroll doesn't move too fast. To speed up the scroll, you can substitute a lower value for 20; to slow it down, you can use a higher value. The next FOR...NEXT loop scrolls the contents back to their original position. It also has a little timing loop nested inside it.

You can use SCROLL to move the contents of a playfield while there are objects on the playfield. SCROLL won't move the objects, it will just move the playfield. However, you might get some strange results if the objects on the playfield are bobs. Since they are actually drawn into the playfield, they can leave a strange "graphics residue" when the playfield scrolls underneath them. If the objects on the playfield are sprites, they won't leave any graphics residue, since sprites aren't part of the the playfield at all.

USING THE PUT STATEMENT FOR SEQUENTIAL ANIMATION

Sprites and bobs are good tools for external animation—Amiga BASIC provides movement and control statements to make the task of animating these objects fairly simple—but they don't have any internal motion. You can position sprites and bobs side-by-side to make a larger object with internal motion, but lining up and

moving individual sprites and bobs can be tricky; it requires keeping track of the location and movement of many separate objects on the playfield at once. Using sequential-animation techniques provided by the PUT statement makes the job easier.

Chapter 6 explains how GET and PUT can save a block of graphics in a variable array and paste it elsewhere in the output window. It also tells how to use GET to store a series of graphics blocks in a two-dimensional array. If you store a series of blocks containing continuous images, you can use PUT to sequentially place those blocks on the screen, creating a moving object with internal and external motion using sequential-animation techniques.

The following program uses GET to store four simple frames of a bird flying. It then uses PUT in two FOR...NEXT loops to cycle through the frames to make the bird look as if it's flying. Figure 13-6 (on the next page) shows the bird's motion.

MakeWindow:

```
SCREEN 1, 320, 200, 2, 1
WINDOW 2, , , 0, 1
```

CreateArray:

```
DIM bird(1440,3)
```

DrawFrames:

```
frame = 0
LINE (160,100) - STEP(-40,-40), 3
LINE (160,100) - STEP(40,-40), 3
GOSUB GetFrame
CLS
```

```
frame = 1
LINE (160,100) - STEP(-50,-15), 3
LINE (160,100) - STEP(50,-15), 3
GOSUB GetFrame
CLS
```

```
frame = 2
LINE (160,100) - STEP(-50,15), 3
LINE (160,100) - STEP(50,15), 3
GOSUB GetFrame
CLS
```

```
frame = 3
LINE (160,100) - STEP(-40,40), 3
LINE (160,100) - STEP(40,40), 3
GOSUB GetFrame
CLS
```

(continued)


```

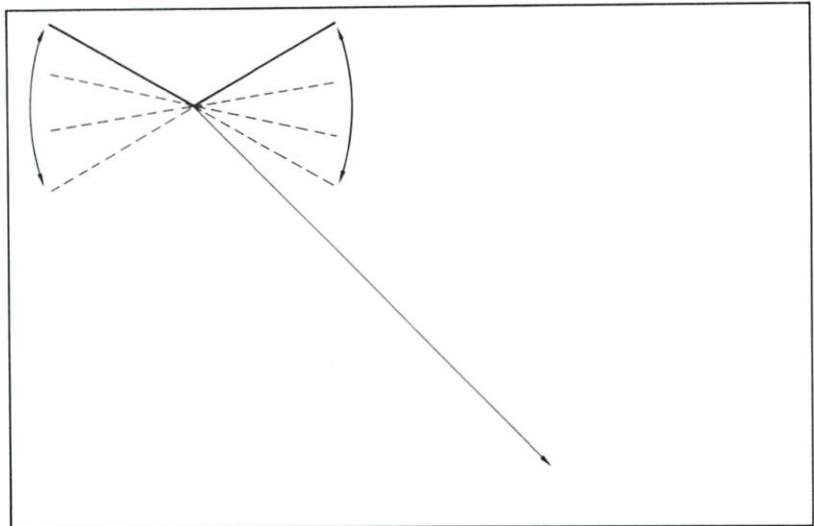
x = 0: y = 0
SequenceFrames:
  FOR i = 0 TO 3
    PUT (x,y), bird(0,i)
    FOR j = 1 TO 200: NEXT j
    PUT (x,y), bird(0,i)
  NEXT i
  FOR i = 2 TO 1 STEP -1
    PUT (x,y), bird(0,i)
    FOR j = 1 TO 200: NEXT j
    PUT (x,y), bird(0,i)
  NEXT i
  x = x + 5: y = y + 5
  GOTO SequenceFrames

GetFrame:
  GET (100,55) - (220,145), bird(0,frame)
  RETURN

```

Figure 13-6.

The program example moves a flapping bird across the playfield.



When you run the program, you'll see a brief flash in the center of the window as it draws the four frames of a bird flying and saves them with the GET statement. The stick bird (it is very simple) will then appear in the upper left corner of the screen, flapping its wings. As it flaps, it flies toward the lower right corner, and passes beyond the output-window border.

To understand how the program works, take a look at its parts. *MakeWindow*: creates a low-resolution, 2-bit-plane screen with a full-sized window. *CreateArray*: creates a two-dimensional array named *bird* to hold the bird frames. The section that follows,

DrawFrames:, is divided into four parts, one for each frame of bird flight. Each part sets a frame number, then draws the bird using two simple LINE statements. It calls the subroutine *GetFrame:* that saves the block containing the bird to the bird array. It then clears the screen for the next part.

The next section, *SequenceFrames:*, runs through the sequence of frames to animate the bird. To initially position the bird, the line before the *SequenceFrames:* section sets the x and y coordinates for placing the PUT block equal to zero.

The two FOR...NEXT loops within the *SequenceFrames:* section flap the bird's wings. The first loop flaps them down, the second flaps them up. You'll notice that there are two identical PUT statements within each loop, separated by a small, nested FOR...NEXT loop. The first PUT statement draws the bird's frame on the screen. The FOR...NEXT loop pauses to keep the image on the screen. The second PUT statement draws the same frame at the same location, effectively erasing the bird's frame from the screen. (Recall that PUT uses the XOR option by default to draw the frame. XORing the same frame twice erases it completely. See Chapter 6 for the details.)

At the end of the two loops is a line that adds 5 to both the x and y coordinates, which changes the PUT block's location for each loop, which makes the bird move down and to the right after each wing-flap cycle. The next line loops the program back to *SequenceFrames:*, which begins the wing-flap cycle over again. There are advantages and disadvantages to this kind of animation. It takes a lot of memory, especially if you use a deep screen, a large frame size, and many frames. It does flicker as you draw the original frames, and it takes a lot of programming work to draw frames that look more realistic than the stick bird you just saw. Nevertheless, it does give you internal motion in an object that can move around the screen. If you want to get rid of the initial drawing flicker, you can draw the frames in one program and save them to disk, then load them for the actual animation program. (You'll have to spend some time getting familiar with the BASIC disk file statements to do this.)

In this chapter you've learned how to limit an object's motion, and how to read an object's location and speed. You've also learned how to handle collisions between objects, how to scroll the playfield, and how to use sequential animation. You should experiment further with the animation statements you learned here—they are the building blocks of animation that, when assembled in new and creative ways, can create some very impressive video animation on your monitor screen.

AFTERWORD

THE FUTURE: AMIGA'S CREATIVE POSSIBILITIES

After you master Deluxe Paint, Deluxe Music, Deluxe Video, Amiga BASIC, and other Amiga graphics, music, and animation tools, will you find yourself up against the wall of ennui with no place left to go? Not likely. The programs that are available now for the Amiga are just the beginning. As more programmers learn to take advantage of the Amiga's unique abilities, you'll have more and more software to play with.

The Amiga has many powers that have not yet been tapped. For example, one of the Amiga's little-used features is the hold and modify (HAM) mode of graphics display, which allows the Amiga to put all of its 4096 colors on the screen at once (with a few limitations). Although some video digitizers can create HAM pictures with incredible shading, there are no graphics programs out now that let you create and print your own HAM pictures from scratch. But as demand increases for added colors, surely programs will be written to satisfy that demand.

Many of the Amiga's animation powers are also little used in today's application programs. For example, the ability to finely scroll a large playfield around the monitor screen isn't used in either Deluxe Video or Aegis Animator. Also unused are the Amiga's system-software routines, which allow a programmer to build an animated character by tying together a series of moving objects to create very realistic motion. Animation programs are new products on the market, so as competition develops, you can expect to see more powerful and refined animation applications for the Amiga.

Musicians are naturally drawn to the Amiga to take advantage of its sound-sampling capabilities, and to use its multitasking power to control and record activities on networks of MIDI instruments. Many of these musicians are also programmers, so in the future you can expect to see programs that let you build very sophisticated instruments using the Amiga's internal sound channels, programs that easily turn your keyboard performances into printed scores, and programs that teach you music notation.

Many of the Amiga's future programs aren't possible to foresee; as users develop new needs and desires, programmers will rise to meet the challenge. Programmers may exercise the internal hardware and Amiga's system software so thoroughly that they'll discover capabilities the Amiga's creators didn't know were there.

No matter how much software and hardware you add to your Amiga, though, the chief goal is to be able to ignore it all and work on your own artistic creations. Any computer, no matter how fascinating in its own right, is just a tool; and the purpose of any tool is to transform its user's ideas into reality as easily and beautifully as possible.



APPENDICES

APPENDIX A: AMIGA BASIC STATEMENT FORMATS

This appendix lists Amiga BASIC graphics, sound, and animation statements and functions along with their formats. They are grouped by function, so you can find the graphics statements together in one section, the sound statements together in another section, and the animation statements together in a third section. Each statement and function has a short description to jog your memory if you've forgotten what the statement or function does.

GRAPHICS

CREATING WINDOWS, SCREENS, AND PALETTES

SCREEN

screen ID number, width, height, depth, resolution mode

This statement creates a screen, setting its size, depth, and resolution.

SCREEN CLOSE *screen ID number*

This statement closes the specified screen and removes it from memory.

WINDOW *window ID number, title, (corner address) - (corner address), window-features number, screen ID number*

This statement creates a window on the screen you specify, using the size, position, and optional-features values you specify for the window.

WINDOW OUTPUT *window ID number*

This statement makes the specified window the output window.

WINDOW CLOSE *window ID number*

This statement closes the specified window and removes it from memory.

PALETTE *color-register number, red strength, green strength, blue strength*

This statement sets the color in the specified color register.

COLOR *foreground-color number, background-color number*

This statement sets the background and foreground colors for subsequent graphics and text commands.

CLS

This statement clears the output window, filling it with the background color.

DRAWING STATEMENTS

PSET (address), color-register number

This statement colors the pixel at the specified address using the color in the specified color register. If no color-register number is specified, it uses the current foreground color.

PRESET (address), color-register number

This statement colors the pixel at the specified address using the color in the specified color register. If no color-register number is specified, it uses the current background color.

*LINE (starting address) - (ending address),
color-register number, box options*

This statement draws a line from the starting address to the ending address using the color in the specified color register. It can also create filled or hollow boxes.

*CIRCLE (center address), radius, color-register number,
arc starting point, arc ending point, aspect*

This statement draws a circle around the center address using the specified radius and the color in the specified color register. It can also draw ovals and arcs.

AREA (address)

This statement sets a point in a polygon to be filled by the AREAFILL statement.

AREAFILL mode number

This statement connects points set by preceding AREA statements and fills in the area they define.

*PAINT (address), paint color-register number, border
color-register number*

This statement fills in an enclosed area with the color in the specified color register, starting at a point specified by the address. PAINT will recognize borders that use the specified border color-register number.

PATTERN line mask, pattern-mask array name

This statement creates line and fill patterns that are used when line-drawing and fill statements put graphics in the output window.

MISCELLANEOUS STATEMENTS AND FUNCTIONS

PRINT expression list

This statement prints the string(s) of characters in the expression list in the output window.

`LOCATE line number, column number`

This statement puts the text cursor at the specified line and column positions.

`CSRLIN`

This function returns the line number of the text cursor's current position.

`POS(0)`

This function returns the column number of the text cursor's current position.

`WINDOW (condition number)`

Depending on the value of the condition number, this function can return the window ID number of the current input or output window, the height, width, and color capacity of the output window, the pixel address of the text cursor in the output window, or pointers to system-software addresses.

`POINT (pixel address)`

This function returns the color-register number of the specified pixel.

`GET (corner address) - (corner address),
array name(index number)`

This statement stores the pixels within the rectangle set by the opposing corner addresses in the specified array.

`PUT (address), array name(index number), merge choice`

This statement puts the pixels stored in the specified array in the output window at the address specified, merging them with the pixels they cover using the specified merge choice.

SOUND

MUSIC

`BEEP`

This statement produces a short beep and flashes the screen.

`SOUND frequency, duration, volume, audio channel`

This statement plays a tone at the specified frequency and volume for the specified duration, using the audio channel specified.

`SOUND WAIT`

This statement queues all subsequent SOUND statements without playing them.

SOUND RESUME

This statement plays all the SOUND statements queued by the SOUND WAIT statement.

WAVE *audio channel, integer-array name*

This statement creates a new waveform in the specified audio channel using the information contained in the specified integer array.

SPEECH

SAY *phoneme-code string, specification-array name*

This statement converts the phoneme code string into speech using the speech characteristics set in the specification array.

TRANSLATE\$(*"text string"*)

This function translates the text string and returns the phoneme codes necessary for the SAY statement.

ANIMATION

CREATING AND MOVING OBJECTS

OBJECT.SHAPE *object id number, object-definition string*

Using this syntax, this statement creates an object in memory from data stored in the object-definition string.

OBJECT.X *object id number, x coordinate*

This statement sets the x coordinate of the specified object's location.

OBJECT.Y *object id number, y coordinate*

This statement sets the y coordinate of the specified object's location.

OBJECT.ON *object id number, object id number, ...*

This statement makes the specified objects (or all objects, if no object ID numbers are specified) visible in the output window.

OBJECT.VX *object id number, x velocity*

This statement sets the x velocity of the specified object.

OBJECT.VY *object id number, y velocity*

This statement sets the y velocity of the specified object.

`OBJECT.START` *object id number, object id number, ...*

This statement sets specified objects (or all objects, if no object ID numbers are specified) in motion.

`OBJECT.AX` *object id number, x acceleration*

This statement sets the amount of acceleration along the x (horizontal) axis for the specified object.

`OBJECT.AY` *object id number, y acceleration*

This statement sets the amount of acceleration along the y (vertical) axis for the specified object.

`OBJECT.STOP` *object id number, object id number, ...*

This statement freezes the motion of the specified objects (or all objects, if no object ID numbers are specified).

`OBJECT.OFF` *object id number, object id number, ...*

This statement makes the specified objects (or all objects, if no object ID numbers are specified) invisible.

`OBJECT.CLOSE` *object id number, object id number, ...*

This statement removes the specified objects (or all objects, if no object ID numbers are specified) from memory.

CONTROLLING OBJECTS

`OBJECT.SHAPE` *duplicate object id number, original object id number*

Using this syntax, this statement creates a duplicate object in memory of an existing object.

`OBJECT.PLANES` *object id number, PlanePick value, PlaneOnOff value*

This statement changes the colors of an existing object.

`OBJECT.CLIP` *(corner address) - (corner address)*

This statement sets an invisible rectangle in the output window that serves as a motion boundary for objects within it.

`OBJECT.X`*(object id number)*

This function returns the x coordinate of the specified object's current location.

`OBJECT.Y`*(object id number)*

This function returns the y coordinate of the specified object's current location.

`OBJECT.VX`*(object id number)*

This function returns the x velocity of the specified object's current motion.

`OBJECT.VY(object id number)`

This function returns the y velocity of the specified object's current motion.

`OBJECT.HIT object id number, type value, hit value`

This statement defines the object type of an object, and defines the types of objects it can hit.

`OBJECT.PRIORITY object id number, priority value`

This statement sets the collision priority for an object, which determines which of two colliding objects passes over the top of the other.

`COLLISION ON`

This statement turns on the collision queue.

`ON COLLISION GOSUB line name/number`

This statement causes Amiga BASIC to jump to the specified program line whenever a collision occurs.

`COLLISION(id number)`

This function returns collision information from the collision queue.

`COLLISION STOP`

This statement suspends the operation of the collision queue.

`COLLISION OFF`

This statement turns off the collision queue and disables ON COLLISION GOSUB.

`SCROLL (corner address) - (corner address), x-shift, y-shift`

This statement shifts the contents of the specified rectangle by the amount specified.

APPENDIX B: COMPANIES MENTIONED IN THIS BOOK

This appendix lists companies mentioned in this book, followed by the company's address and a list of their products discussed in this book.

Aegis Development
2210 Wilshire #277
Santa Monica, CA 90403
*Products: Aegis Animator,
Aegis Images*

A-Squared Systems
10 Skyway Lane
Oakland, CA 94619
Product: Amiga LIVE! framegrabber

Casio, Incorporated
15 Gardner Road
Fairfield, NJ 07006
*Products: AS-20 speaker, CZ-101
synthesizer, CZ-1000 synthesizer*

Commodore-Amiga, Incorporated
983 University Avenue #D
Los Gatos, CA 95030
*Products: Amiga 1000 computer,
Amiga 1300 Genlock Board, Amiga
BASIC, AmigaDOS, Amiga
Kickstart, Amiga Workbench,
Graphicraft, Textcraft*

Electronic Arts
1820 Gateway Drive
San Mateo, CA 94404
*Products: DeluxeMusic, DeluxePaint,
DeluxeVideo, Instant Music*

Epson America, Incorporated
2780 Lomita Boulevard
Torrance, CA 90505
Product: JX-80 printer

The Micro Forge
398 Grant Street, S.E.
Atlanta, GA 30312-2227
Product: RAM cards

Mimetics Corporation
P.O. Box 60238 Station A
Palo Alto, CA 94306
*Products: SoundScape,
SoundScape sound sampler*

Okidata
532 Fellowship Road
Mt. Laurel, NJ 08054
Product: Okimate 20 printer

Pioneer Video, Incorporated
200 West Grand Avenue
Montvale, NJ 07645
*Products: CLD-900 LaserDisc
player, IU-04 LaserDisc computer
interface*

Sony Corporation of America
Sony Drive
Park Ridge, NJ 07656
Product: KV-1311 video monitor

Teac Corporation of America
7733 Telegraph Road
Montebello, CA 90640
*Product: TASCAM Ministudio
Porta One cassette recorder*

Video Vision Associates Limited
7 Waverly Place
Madison, NJ 07940
*Product: Space Archive
LaserDiscs*

Xerox Corporation
901 Page Avenue
Fremont, CA 94538
*Products: Diablo C-150 inkjet
printer, Xerox 4020 inkjet printer*

INDEX

Page numbers for illustrations are in italics

A

- Aegis Animator, 264, 289–90, 351
- Amiga BASIC, 23, 41, 45, 351
- Amiga BASIC, animation, 296–325.
See also Object Editor
 - controlling motion, 328–49 (*see also* Appendix A; *specific statements and functions*)
 - creating a motion boundary, 328–29 (*see also* OBJECT CLIP statement)
 - drawers, storing files in, 308–9
 - gels, 264
 - handling object collisions, 333–44 (*see also* Appendix A; *specific statements and functions*)
 - placing and moving objects, 306–19 (*see also specific statements*)
 - playfields, 263–67, 301
 - reading an object's location and velocity, 330–33, 333 (*see also specific object functions*)
 - scrolling, 263
 - tricks, 344–49
- Amiga BASIC, graphics, 86–169. *See also* Appendix A; Area pattern; Color palette; Screen; Windows
 - color registers, 107–12
 - image creation, 114–41 (*see also* Graphics cursor and pixel addressing)
 - list and output windows, 86–89
 - miscellaneous graphics and functions, 144–69 (*see also* Appendix A; Character addressing; Fonts; Graphics, copying and pasting; Graphics, interactive)
 - Workbench screen window, 92
- Amiga BASIC, sound, 222–48. *See also* Appendix A, *specific SOUND statements*; Speech, generation
 - musical scale array, 233–35
 - playing a musical score, 235–38
 - synchronization, 227–28
- AmigaDOS (Disk Operating System), 21, 23
 - CLI commands, 23
 - Exec, 19–20
 - RAM, 21
- Amplifier, 180
- Amplitude (loudness), 176, 186, 188
- Animation, 252–325. *See also* Deluxe Video
 - fundamentals, 252–62 (*see also* Illusion of motion; Perspective)
 - IFF standard, 291
 - interactive, 262, 267–68
 - internal motion, 265–67
- Anti-aliasing, 39
- Area pattern, 138–41
 - applying an area pattern, 139–41
 - putting the area pattern mask into an integer array, 138–39
- AREA statement, 129
- AREAFILL statement, 130–32
 - filling a shape with a pattern, 132
- Array, musical scale, 233–35
 - traditional music notation, 234
- Array, PUT and GET variable, 159–64. *See also* PUT and GET statements
 - calculating size, 161–62
 - copying several equal-sized graphics blocks in one array, 162
 - copying several unequal-sized graphics blocks in the same array, 163
 - storing a graphics block, 162
 - storing graphics data, 160–61, 161
- Array, specification, 242–55. *See also* SAY statement
 - choosing the channel assignment, 244–45
 - choosing inflection or monotone, 243
 - choosing multiple SAY options, 245
 - choosing synchronous or asynchronous speech, 245
 - choosing voice gender, 243
 - setting base pitch, 242
 - setting sampling frequency, 243–44
 - setting speaking rate, 243
 - setting volume, 244
- Array, waveform. *See* Waveform table
- ASCII (American Standard Code for Information Interchange), 193

B

- BEEP statement, 222
 - waveform of, 222
- Binary line mask, 136–37
- Bit planes, 57, 95–96
- Blend mode, 65, 70
 - creation of rainclouds, 71
- Blitter, 48
- Bob, 301
- Brush modes, Deluxe Paint, 63–72
 - blend, 65
 - blend to create rainclouds, 70
 - color, 63–64
 - cycle, 65
 - object, 63
 - object and color to outline figures, 66–67
 - replace, 64
 - shade, 64
 - shade to create shadow, 71–72
 - smear, 64
 - smear to create sandy background, 68
 - text brush to create outlined text, 67

C

- Casio CZ-101 synthesizer, 218
- Casio CZ-1000 synthesizer, 218
- Character addressing, 144–45
 - line and column numbering, 145
- CIRCLE statement, 124–29, 125
 - circle measured in radians, 127
 - creating arcs, 126, 126
 - creating circles, 125
 - creating ovals, 127–29
- CLI (Command Line Interpreter), 22, 74
- CLS Statement, 119
- COLLISION() function, 339–41
 - collision queue, 339, 340
- COLLISION OFF statement, 344
- COLLISION ON statement, 339–41
- COLLISION STOP statement, 343
- Color, 28–31
 - HIS (Hue, Intensity, Saturation) creation, 30–31
 - primary, 28–29
 - RGB creation, 29
- Color mode, 63–64, 66–67
- Color palette, 46–47, 107–12
 - design, 57–59
 - HAM (Hold and Modify), 47
 - palette window, 58
 - picture using HAM mode for shading, 47
 - range, 59
- COLOR statement, 108–9
 - changing background color, 109
 - changing foreground color, 109
 - with PRINT statement, 151–52
 - storing color, 110

- Commodore-Amiga, 79, 80
- Component motion, 266–67
- Console, 7, 7
 - custom chips, 9
 - internal disk drive, 10–11
 - memory chips, 10
 - motherboard, 8
 - Motorola 68000 microprocessor, 9
 - power supply, 7–8
- Creating precise drawings, 60–63
 - coordinates command and brush position, 60
 - grid alignment, 61–63
 - SHIFT key and drawing straight lines, 60–61
- Cycle mode, 65

D

- Deluxe Music, 23, 196–213
 - capabilities, 198–206
 - recording, 209–12
 - special effects using Tascam Ministudio Porta One recorder, 210–11
 - synchronizing scores, 211–12
 - tips, 208–9
 - use with MIDI, 206
- Deluxe Paint, 23, 41, 45, 48, 49, 54–83, 263, 265. *See also* Brush modes; Color palette
 - choosing screen resolution and depth, 56–57
 - creating background and matching objects for animation, 271–74
 - creating precise drawings and color-cycle animation, 274–79
 - creating sequenced drawings for animation, 279–85
 - creating a work disk, 55–56
 - photographing, 78
 - printing Deluxe Paint pictures, 74–78
 - tricks, 73–74
- Deluxe Video, 23, 45, 264
 - and Deluxe Paint, 271–85 (*see also* Deluxe Paint)
 - blitter, 270, 275
 - color cycle, 274–79, 276, 277, 279
 - library disks, 290
 - making scripts readable, 286, 287
 - recording onto videotape, 287–89
 - sequential drawing, 279–85, 280, 281, 282, 283
 - synchronizing effects with sound, 285
- Deluxe Video Framer, 279, 280, 282
- Deluxe Video Maker, 279, 283, 284
- Denise (chip), 264
- Direct Memory Access (DMA), 10–11
- Drawing pictures, 48
 - and Agnus chip, 48
 - blitter, 48

Duplicating objects. *See*
OBJECT.SHAPE statement
Duration, 186, 187, 235
and SOUND statement, 225

E

Electronic Arts, 212, 270, 290
Exec, 19–20
and Kickstart disk, 19–20
and ROM, 19
External disk drive, 16

F

Firmware, 10
Fonts, using different sizes, 145–46
Frequency (of sounds), 176
and SOUND statement, 223–25
Functions. *See also* Appendix A and
specific functions

G

Gels, 264–67
bobs, 265
sprites, 264–65, 300
GET statement, 159, 167–69
Graphicraft, 263
Graphics, copying and pasting, 159–
69 *See also* Array, PUT and
GET variable; GET
statement; PUT statement
examples using GET and PUT,
167–69
Graphics cursor and pixel
addressing, 114–41. *See also*
specific graphics statements;
Multi-sided figure creation
absolute addressing, 114–15
relative addressing, 115–16
window boundaries, 117–18, 117
Graphics, features, 41–52. *See also*
Color palette; Drawing
pictures; Mixing two
pictures; Picture detail;
Scrolling; Text, creation
Graphics, interactive, 153–58. *See*
also POINT function;
WINDOW() function

H

Hardware, 4–17, 4. *See also*
Console; Keyboard; Monitors;
Mouse
manufacturers (mentioned in text),
360–61
Hardware, animation, 291–94
Amiga 1300 Genlock board, 292
external RAM cards, 291–92
laser disc players, 293–94
Hardware, graphics, 80–83. *See also*
Monitors; Printers
Amiga Live! framegrabber, 83

Hardware, sound
speakers, 215–16
stereo, 15
Hawkins, Trip, xi
Hexadecimal numbers, 136–37

I

Illusion of motion, 252–57, 253
background, 254–55
moving objects, 255–57
Infinite loop, 122
Interlacing, 94
Intuition, 21, 45
user-interface routines, 21

K

Keyboard, 5–6, 6

L

Libraries and devices, 20, 263
Line pattern, 136–38
applying a line pattern, 137
converting a binary line mask into
a hexadecimal number, 136–
37
LINE statement, 122–24
creating boxes, 123–24, 124
creating lines, 122–23, 123
LOCATE statement, 148–49

M

Memory Chips, 10. *See also* RAM;
ROM
MIDI (Musical Instrument Digital
Interface), 190–93, 192
adaptor, 218
creating a MIDI ring, 192
messages, 193
physical standard, 191–92
Mixers, 189
Mixing two pictures, 51–52, 51
Modem, 13, 17, 193
Monitors, 5, 31–41, 80–81
cathode-ray tube, 32
color, 34–35 35
composite, 5
CRT electron gun, 32
monochrome, 31–34
phosphor dots of a color monitor,
34
ports, 5
raster scanning, 33–34, 33
RGB, 5, 80
television, 5
transferring a video image to other
media, 35–41 (*see also*
Photographing the monitor
screen; Printing video
images; Videotape)
Mouse, 6

- Multi-sided figure creation, 129–32, 132. *See also* AREA statement; AREAFILL statement
 - Musical score, playing. *See* Amiga BASIC, sound
- N**
- NTSC (National Television Systems Committee), 14
- O**
- Object Editor, 296–301
 - drawing an object, 299–301, 300
 - modification of, 297–99
 - Object mode, 63, 66–67
 - OBJECT.AX statement, 316–17
 - OBJECT.AY statement, 316–17
 - OBJECT.CLIP statement, 329, 330
 - OBJECT.CLOSE statement, 318–19
 - OBJECT.HIT statement, 334–37, 337
 - OBJECT.OFF statement, 317–18
 - OBJECT.ON statement, 311–12, 312
 - OBJECT.PLANES statement
 - to change colors, 320
 - example, 325
 - setting PlanePick and PlaneOnOff values, 323–25
 - OBJECT.PRIORITY statement, 337–38
 - OBJECT.SHAPE statement, 307–8
 - to duplicate objects, 319–20
 - OBJECT.START statement, 315
 - OBJECT.STOP statement, 317
 - OBJECT.VX() function, 331–33
 - OBJECT.VX statement, 312–15, 313, 314
 - OBJECT.VY() function, 331–33
 - OBJECT.VY statement, 312–15, 313, 314
 - OBJECT.X() function, 330–31
 - OBJECT.X statement, 309–11, 310
 - OBJECT.Y() function, 330–31
 - OBJECT.Y statement, 309–11, 310
 - ON COLLISION GOSUB statement, 338–39
 - Open machine, 12
 - Output window, 105–6
- P**
- PAINT statement, 133–34
 - to fill in a circle, 134, 135
 - PALETTE statement, 109–112
 - creating different palettes for different scenes, 111–12
 - 16 primary color strengths and corresponding PALETTE values, 110
 - two different screens with two unique screen palettes, 112
 - Palettes. *See* Color palette
 - PATTERN statement, 135–41, 140. *See also* Area pattern; Line pattern
 - applying a line pattern, 137, 137
 - changing pattern colors, 141
 - conversion of binary line mask into a hexadecimal number, 136–37
 - and creation of line pattern, 136
 - Peripherals, 11, 14–17
 - external disk drives, 16–17
 - external RAM, 17
 - impact dot-matrix printers, 15
 - ink-jet printers, 15
 - laser printers, 16
 - letter-quality printers, 15
 - thermal-transfer printers, 15
 - Perspective, 257–62
 - three-dimensional, 258–60, 259, 260
 - two-and-a-half-dimensional, 260–62, 261
 - two-dimensional, 257–58, 257
 - Photographing the monitor screen, 36–37
 - disadvantages, 37
 - Picture detail, 42–45. *See also* Resolution
 - pixels, 42
 - Pioneer CLD-900, 293
 - Pioneer IU-04 computer interface, 293
 - Pitch, 186, 188, 235
 - Pixel addressing. *See* Graphics cursor and pixel addressing
 - Playfield, 271–75
 - Playfield, animation, 301–4, 305
 - choosing colors, 303–4
 - screen depth, 302–3
 - screen resolution, 302
 - POINT function, 158
 - Ports and connectors, 11–14. *See also* Peripherals
 - audio, 13–14
 - on back of console, 12
 - controller, 12
 - disk-drive, 13
 - expansion connector, 12
 - internal RAM connector, 11
 - keyboard, 13
 - MIDI, 191
 - parallel, 13
 - RGB, 14
 - on right and front side of console, 11
 - serial, 13, 192
 - TV modulator, 14
 - video, 14
 - Preferences, 52, 74–8
 - Change Printer screen, 74–76, 78, 75, 76
 - Graphic Select screen, 76–78, 77
 - PRESET statement, 120–22

PRINT statement, 147
 with COLOR statement, 151–52, 152

Printers, 15, 16, 81–82
 Diablo C-150 and Xerox 4020, 82
 Epson JX-80, 82
 Okimate 20, 81

Printing
 printer driver, 52, 74

Printing Deluxe Paint pictures, 74–78. *See also* Preferences

Printing video images
 converting color to black and white, 39–40, 40
 fidelity, 37–38
 printing in color, 40–41

PSET statement, 119–22
 with trigonometric command, 121

PUT statement, 164–65
 array name and index, 164–65
 merge choices, 165–67
 sequential animation, 346–49, 348
 setting PUT address, 164

R

RAM (Random Access Memory), 10
 and animation storage, 291–92
 and bit planes, 57
 and fonts, 49
 and resolution, 44
 and sequencers, 181
 and window refreshing, 102–3
 and Workbench, 21

Recording, animation, 287–89
 quality, 288

Recording, Deluxe Music scores, 209–12

Recording, sound, 188–90

Replace mode, 64

Resolution
 affect on memory, 44–45
 animation, 302
 anti-aliasing, 39, 39
 depth, 56–57
 font size, 146, 147
 "jaggies," 39, 38
 mixing, 45, 45
 picture using high-resolution pixel, 44
 picture using low-resolution pixel, 43
 SCREEN statement, 93–94
 Workbench display using 640-by-200 resolution screen, 42

RGB monitor, 5, 80–81
 analog, 80–81
 controls of the Preferences program, 29
 digital, 80–81

ROM (Read Only Memory), 10, 19

S

SAY statement, 241–47. *See also*
 Array, specification
 examples, 246–47
 phoneme codes, 241–42

Screen (horizontal areas), 45

SCREEN CLOSE statement, 98

SCREEN statement, 93–98
 conserving memory, 96
 depth, 95–96
 examples, 96–97
 five simultaneous screens, 97, 97
 ID number, 93
 memory requirements, 96
 mode 1 screen, 95, 95
 resolution mode, 93–94
 width and height, 94–95

Screens, 90–91, 93–98. *See also*
 SCREEN CLOSE statement;
 SCREEN statement
 title bar, 90, 91

SCROLL statement, 342
 scrolling the playfield, 344, 345

Scrolling, 50

Sequenced drawings, 266

Sequencer, 187

Serial printer, 13

Shade mode, 64, 71–72
 creation of shadow, 72

Sizing gadget, 118

Smear mode, 64, 68–69
 creation of sandy background, 69

Software, 17–23, 19. *See also*
 AmigaDOS; CLI (Command Line Interpreter); Exec; Intuition; Libraries and devices; Workbench
 manufacturers (mentioned in text), 360–61

Software, animation. *See* Aegis Animator; Amiga BASIC, animation; Deluxe Video

Software, graphics, 79–80. *See also*
 Amiga BASIC; Deluxe Paint
 Aegis Images, 79
 Graphicraft, 79
 IFF (Interchange File Format) Graphics Standard, 80

Software, sound. *See* Amiga BASIC, sound; Deluxe Music
 IFF music standard, 214
 Instant Music, 212

Sound. *See also* MIDI; Recording;
 Sounds, electronic;
 Synthesized music;
 Synthesized speech
 characteristics of, 176–78
 and hearing, 174–76
 sampled, 185–87
 samplers, 217
 sine, square, and sawtooth waves, 177

Sound, synchronization. *See* Amiga
 BASIC, sound; SOUND
 RESUME statement; SOUND
 WAIT statement
 SOUND RESUME statement, 228
 SOUND statement, 223–28
 choosing audio channel, 226
 setting duration, 225
 setting frequency, 223–25, 224
 setting volume, 226
 SOUND WAIT statement, 227–28
 Sounds, electronic, 179–81
 Speakers, 179–80
 Speech, generation, 238–47 (*see*
 also SAY statement;
 TRANSLATE\$() function;
 Statements. *See* Appendix A and
 specific statements
 Stereo, 180–81
 Synthesized music, 181–87. *See also*
 Sequencer; Synthesizers
 Synthesized speech, 188
 Synthesizers
 analog, 182
 digital, 182–87
 digital-to-analog converter (D-to-A
 converter), 184

T

Tape recorders, 189
 Text, creation, 48–50, 49
 and background, 50
 font, 48–49
 Text cursor
 locating with CSRLIN and POS(0)
 functions, 150–51
 Textcraft, 49
 Timbre, 177, 186, 187, 188, 222
 Transformer IBM emulator, 16
 TRANSLATE\$() function, 239–41
 alternate spellings for correct
 punctuation, 240–41
 using punctuation with, 240
 Trigonometry, 121–22

V

Video-signal converter, 12
 Video Vision Associates Space
 Archive laser disc, 294
 Videotape, 36
 Voltage-controlled oscillator (VCO),
 182
 Volume, 235
 and SOUND statement, 226

W

WAVE statement, 229–33. *See also*
 Waveform table
 Waveform, 177, 183–87, 222, 183,
 230, 231, 232
 Waveform table, 229–33, 238
 assigning to audio channel, 232–
 33
 calculation, 231–32
 creation of, 229–30
 reading data of, into waveform
 array, 230–31
 WINDOW CLOSE statement, 106–7
 WINDOW() function, 153–57
 examples, 154–57, 155, 157, 158
 table, 154
 WINDOW OUTPUT statement, 106
 WINDOW statement, 98–105
 examples, 104–5
 features, 101–3
 ID number, 99
 opposing corner addresses, 99
 pixel-addressing system, 100
 screen ID number, 103–4
 window in upper corner of screen,
 101
 window with all possible features,
 103
 Windows, 91–92, 98–107. *See also*
 Output window; WINDOW
 CLOSE statement; WINDOW
 OUTPUT statement;
 WINDOW statement
 Workbench, 21–22, 48, 52, 22
 and Intuition, 22

About the Author

A classical oboist by training, Michael Boom became involved with computers in 1980, when he purchased an Atari 800. He is the author of *Understanding Atari Graphics*, *How to Use Atari Computers*, and *How to Use the Commodore 64*, all published as Alfred Handy Guides. He has also written for *Compute!* magazine. For the past two years, Michael has worked as a consultant in the microcomputer field, including a year as the music consultant for Commodore-Amiga. Michael Boom currently lives in Oakland, California.

The manuscript for this book was prepared and submitted to Microsoft Press in electronic form. Text files were processed and formatted using Microsoft Word.

Cover design by Ted Mader and Associates
Interior text design by Woodfin Design
Illustrations by Rick Bourgojn
Polyscope images courtesy of Dan Silva
Principal typographer: Russell Steele
Principal artist: Becky Johnson

The screen photographs were created on the Commodore Amiga and photographed by Nick Gregoric.

Text composition by Microsoft Press in Serifa 45 with display in Serifa 65, using the CCI-400 composition system and the Mergenthaler Linotron 202 digital phototypesetter.

THE AMIGA

Images,
Sounds, and
Animation
on the
Commodore®
Amiga™

Michael Boom

Just as the Commodore Amiga dazzled the computer world when it was introduced, *The Amiga* by Michael Boom will spur the creative energies of everyone using or planning to purchase this incredible machine.

If you are a graphic artist, an audio/video specialist, or a computer enthusiast with some BASIC programming experience, get ready to produce amazing results with your 512K Amiga that go far beyond the beginner's level.

- Explore the possibilities of developing sophisticated visual images and learn how to enhance the power of the Object Editor.
- Reproduce real and synthesized sound with the Amiga's built-in synthesizer
- Create animated sequences and record them on vidcotape

In addition to information about Amiga BASIC, you'll learn advanced techniques for using some long-awaited software — Deluxe Paint, Deluxe Music, and Deluxe Video. *The Amiga* — your guide to mining the rich artistic depths of this fantastic machine.

USA \$19.95
UK £16.95
AUST. \$29.95
(recommended)

ISBN 0-914845-62-4