3D Logo™

A Multimedia Logo for the Apple IIGS

Mike Westerfield

Byte Works®, Inc.
4700 Irving Blvd. NW, Suite 207
Albuquerque, NM 87114
(505) 898-8183

Master Set 1.0.0.0

Limited Warranty - Subject to the below stated limitations, Byte Works, Inc. hereby warrants that the programs contained in this unit will load and run on the standard manufacturer's configuration for the computer listed for a period of ninety (90) days from date of purchase. Except for such warranty, this product is supplied on an "as is" basis without warranty as to merchantability or its fitness for any particular purpose. The limits of warranty extend only to the original purchaser.

Neither Byte Works, Inc. nor the authors of this program are liable or responsible to the purchaser and/or user for loss or damage caused, or alleged to be caused, directly or indirectly by this software and its attendant documentation, including (but not limited to) interruption of service, loss of business, or anticipatory profits.

To obtain the warranty offered, the enclosed purchaser registration card must be completed and returned to the Byte Works, Inc. within ten (10) days of purchase.

Important Notice - This is a fully copyrighted work and as such is protected under copyright laws of the United States of America. According to these laws, consumers of copywritten material may make copies for their personal use only. Duplication for any purpose whatsoever would constitute infringement of copyright laws and the offender would be liable to civil damages of up to $50,000 in addition to actual damages, plus criminal penalties of up to one year imprisonment and/or a $10,000 file.

This product is sold for use on a *single computer* at a *single location*. Contact the publisher for information regarding licensing for use at multiple-workstation or multiple-computer installations.

Apple Computer, Inc. MAKES NO WARRANTIES, EITHER EXPRESSED OR IMPLIED, REGARDING THE ENCLOSED COMPUTER SOFTWARE PACKAGE, ITS MERCHANTABILITY OR ITS FITNESS FOR ANY PARTICULAR PURPOSE. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY PROVIDES YOU WITH SPECIFIC LEGAL RIGHTS. THERE MAY BE OTHER RIGHTS THAT YOU MAY HAVE WHICH VARY FROM STATE TO STATE.

GS/OS is a copyrighted program of Apple Computer, Inc. licensed to Byte Works, Inc. to distribute for use only in combination with 3D Logo. Apple software shall not be copied onto another diskette (except for archive purpose) or into memory unless as part of the execution of 3D Logo. When 3D Logo has completed execution Apple Software shall not be used by any other program.

Apple is a registered trademark of Apple Computer, Inc.

Table of Contents

Chapter 1 – Getting the Most from 3D Logo

What's in This Book

Like most books that come with a program, this one serves three purposes.

- This book shows you how to install 3D Logo on your computer.
- This book shows you how to use 3D Logo.
- This book includes a reference section that is a catalog of all of the commands in 3D Logo.

This section gives you a quick overview of the book, as well as a detailed overview of this chapter. If you read this section carefully, you'll learn what you can safely skip, and what you really need to read. If you're an experienced computer user, especially if you already know Logo, you'll end up skipping a lot.

This book is divided into three major sections.

- This chapter tells you how to install 3D Logo, where to go to find others who are using 3D Logo, and how to get help if you need it.
- Chapters 2 through 6 are a tutorial introduction to the Logo language in general, and 3D Logo in particular. Learning by example and exploration, you'll write real, working Logo programs. You'll learn how the Logo language works, and how you can use it to draw pictures, create 3D images, make movies, talk, and even write desktop programs that run from Apple's Finder.
- Chapters 7 and 8 are the reference manual for 3D Logo. These chapters are a dictionary of the commands you can use in 3D Logo, many of which are not used in the introductory chapters. There are sample subroutines, programs, or short commands you can type that show how each command is used.

This chapter helps you set up 3D Logo. You'll learn how to install 3D Logo on your computer, how to back up the disks, what to do if you have questions or problems, and where to go to find out more about Logo. I'd suggest skimming most of the chapter. Here's a quick synopsis so you know what you can skip and what you need to read:

Backing Up 3D Logo

If you're an experienced computer user, you already know how to copy floppy disks to make a backup, and even why you should make a backup. If not, be sure you read this section carefully.

Installing 3D Logo on a Hard Disk

Hard disks run faster and are more reliable than floppy disks, so if you have a hard drive, you'll want to put 3D Logo on your hard drive right away. This section tells you how.

Using 3D Logo With Floppy Disks
3D Logo is not a huge program. It will run very well from floppy disks. This section gives some hints to get the most from 3D Logo, based on the number and kind of floppy disk drives you have.

Using 3D Logo from a Network
If you are using 3D Logo in a school with a network, you'll want to install 3D Logo on your network. This section tells you how.

Getting Answers to Questions
If you have questions or problems, you may want to talk with us or to other people who are using 3D Logo. This section tells you how to get help and where to swap ideas.

Finding Out More About Logo
Logo is a popular language, but most bookstores don't carry Logo books. This section lists a few, and tells you how to find more.

Backing Up 3D Logo

There are two disks that come with 3D Logo. Floppy disks are a very reliable way to store information, but accidents do happen. If your floppy disk gets too hot, or gets too close to a magnet, the program might be partially erased – and if that happens, it needs to be re-recorded. It's also possible you could loose a disk.

Of course, if you need to re-record a disk, you need to have more than one copy. That's what backups are for.

We suggest making one backup copy of your floppy disks. You should also have a working copy, generally the one on your hard disk.

If you have a disk copy program you already know how to use, go ahead. The 3D Logo disks are not copy protected, so you can copy them with any copy program.

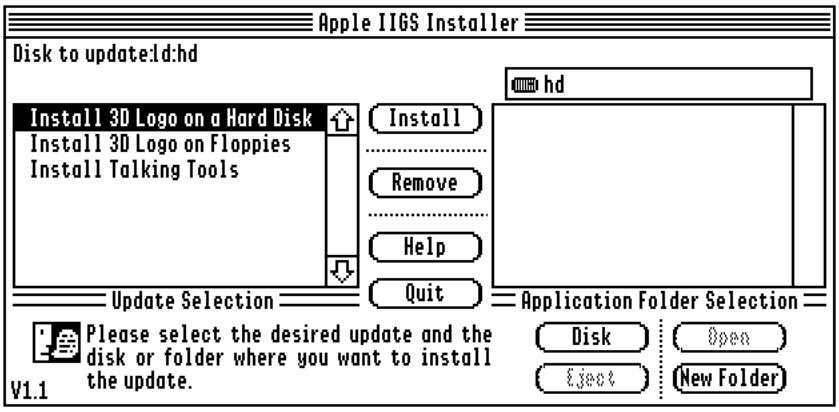
Here's how to make backup copies from the Finder. The Finder is the program supplied by Apple Computer that shows disks, lets you copy files, and lets you run programs.

- Get into Apple's Finder. One way to do that is to boot from the System Disk that comes with 3D Logo.
- Insert a blank floppy disk into the computer. The Finder will ask you to initialize the disk. Initialize it, using the default name of Untitled.
- Eject the disk by pulling down the Disk menu and selecting Eject. The disk has to be selected before you can eject it. If it doesn't eject, click one time on the disk icon to select the disk, then try again.
- Insert the Logo system disk into the disk drive.

- Drag the disk icon for the disk you want to copy to the disk you just initialized. The Finder will double-check before copying the disk, and will ask you to swap the disks a few times. (Of course, if you have two 3.5 inch floppy disk drives, you can leave both disks in drives and avoid the swapping.)
- Click on the name of the new disk Type the name of the disk you copied, since the name of the disk can be important. Press RETURN to finish naming the disk.
- Eject the new disk by dragging it to the trash can.
- Set the write-protect tab so the disk can't be accidentally erased, and store the disk in a safe place.
- Repeat this process for the second 3D Logo disk.

Installing 3D Logo on a Hard Disk

Insert the disk called Program Disk and run the Installer. This program will copy 3D Logo and all of it's sample files to your hard disk. You'll see a screen that looks like this:



Select "Install 3D Logo on a Hard Disk" from the list on the left, and pick out a folder from the list on the right. (Be sure to click on the installer script once, even if it looks selected already.) If you want to create a new folder, use the list on the right to select a place for the new folder, then click on New Folder. Once you have selected the script and a destination folder, click on the Install button. The Installer will copy Logo3D.Sys16 to your hard disk, as well as all of the samples.

If you have the program Talking Tools from the Byte Works, Inc., 3D Logo can talk. To install the tools to 3D Logo can talk, select the script "Install Talking Tools" from the left list, then click on Install. This copies four tools to your system folder, so it doesn't matter what folder is selected from the right list. Installing these tools changes your system folder, so you'll have to reboot your computer when you finish.

▲ **Warning** 3D Logo requires System Disk 6.0 or later. If you have an older version of Apple's operating system, you will need to update your operating system or boot from the System Disk that comes with 3D Logo.

If you don't have the latest System Disk, and would like to get a copy, check first with your local dealer, your user's group, and any online services you have access to. You may be able to get a copy of the latest System Disks free. If all else fails, or if you want to be sure you're getting the complete set of disks and release notes, you can buy them from Resource Central.

Resource Central
6339 West 110th
Overland Park, KS 66211
(913) 469-6502 ▲

The file name for 3D Logo is Logo3D.Sys16. Run this program to start 3D Logo.

Using 3D Logo With Floppy Disks

You can boot the System Disk that comes with 3D Logo and run logo from the Program Disk, and you might want to do just that. There are some other options that will probably work better for you, though.

As our disks ship, you can boot the System Disk, which starts Apple's Finder. From there you can run 3D Logo by double-clicking on Logo3D.Sys16. If you only have one 3.5 inch floppy disk drive, you'll have to eject the system disk and insert the program disk before you can run the program, then swap the disks once as Logo loads tools from the system disk.

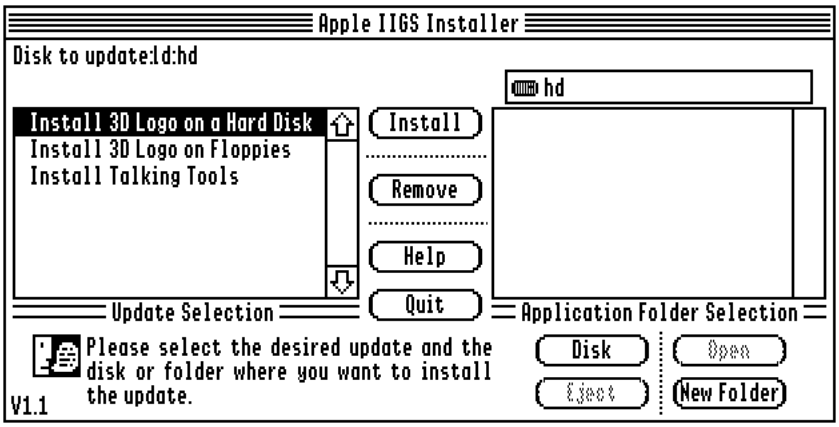
There are two problems with running 3D Logo this way. First, it takes time to run the Finder, especially from floppy disks. If you start in 3D Logo instead of in the Finder, you can start using 3D Logo faster. Second, if you are using one 3.5 inch floppy disk drive, you have to swap disks. It would be easier if everything would fit on a single disk, and it can.

To set up a one-disk system, start by initializing a disk from the Finder, naming the disk OneDiskLogo. This disk will eventually contain the system files needed to boot your computer and the 3D Logo program. You will need a second disk, too, for the Logo samples and the Logo programs you write. If you have two 3.5 inch floppy disk drives, the second floppy disk should be a 3.5 inch floppy; if you have one 3.5 inch floppy disk drive and one 5.25 inch floppy disk drive, make the second disk a 5.25 inch floppy disk. Initialize your second disk as OneDiskSamples.

▲ **Warning** You must use the names OneDiskLogo and OneDiskSamples for your floppy disks, or the Installer won't be able to find

your disks. You can change the names of the disks later, after you finish with the Installer. ▲

Next, insert the disk called Program Disk and run the Installer. This program will copy 3D Logo and all of it's files to your hard disk. You'll see a screen that looks like this:



Select "Install 3D Logo on Floppies" from the list on the left, then click on the Install button. The Installer will copy all of the files you need to use 3D Logo to your floppy disks.

Your floppy disk has all of the files you need to boot and run 3D Logo, but it doesn't have the run-time module or samples. Your second floppy disk, OneDiskSamples, has the sample files that we'll use in the tutorial chapters.

To run 3D Logo, put the disk OneDiskLogo in your boot drive and boot your computer. You should save files to a different disk – either OneDiskSamples or some other floppy disk you want to use for Logo programs.

Using Talking Tools from Floppies

If you have the program Talking Tools from the Byte Works, Inc., 3D Logo can talk. Unfortunately, there isn't enough room on a single 3.5" floppy disk to hold the system files, 3D Logo and the talking tools, so you will have to use two 3.5" floppy disks and swap disks as you boot.

The first step is to create a working copy of the Logo System Disk. This copy will be the one 3D Logo will boot from, and the one the speech tools are actually installed on. Once you make the working copy of the system disk, restart your computer by booting from the new disk.

To install the tools so 3D Logo can talk, run the Installer and select the script "Install Talking Tools on a Floppy" from the left list, then click on Install. This copies four tools to your copy of the 3D Logo System Disk.

Using 3D Logo from a Network

You can install 3D logo on a network using the same script that installs 3D Logo on a hard disk. Once installed, you can use 3D Logo just like you use any other program.

There is one thing you need to be aware of when using 3D Logo from a network: 3D Logo requires System 6.0 or better. If you are running an older version of Apple's system software, you will need to update your system software or boot from floppy disks.

Getting Answers to Questions

Eventually, you'll probably want to talk with others about Logo.

If you need technical assistance – anything from a bad disk to not understanding the manual to reporting a bug – you should contact the publisher and ask for technical assistance. The publisher occasionally adds new technical support channels or changes hours or phone numbers. For the latest times, numbers, online services and mailing address, read the file Tech.Support, which you can find on the Program Disk.

If you would like to talk to others about Logo in general or 3D Logo in particular, we suggest one of the major online services, like GEnie or America Online. You'll probably find a topic to discuss 3D Logo already there, and can join in the discussion. As this manual goes to press, the publisher sponsors discussions on America Online and GEnie. You can join the discussion on America Online using keyword ByteWorks. From GEnie, use A2Pro, then look in category 36 of the bulletin board.

Finding Out More About Logo

This manual has a short introduction to the Logo language, but there are a lot of other good books about Logo. Unfortunately, very few bookstores actually have Logo books on their shelves. That doesn't mean there aren't any good ones around, but you have to ask for them. For a complete list of Logo books, ask any book seller for the Subject volume of Books in Print, and look under Logo. There are several columns of Logo books listed there. Most book sellers will be happy to order almost any of the books you find there.

Here's a few of the more prominent Logo books.

Exploring Language with Logo
E. Paul Goldenberg and Wallace Feurzeig
The MIT Press, 1987
This book is about human languages. You'll learn about the Logo language as you write programs that analyze English – and even programs that write original poems and prose.

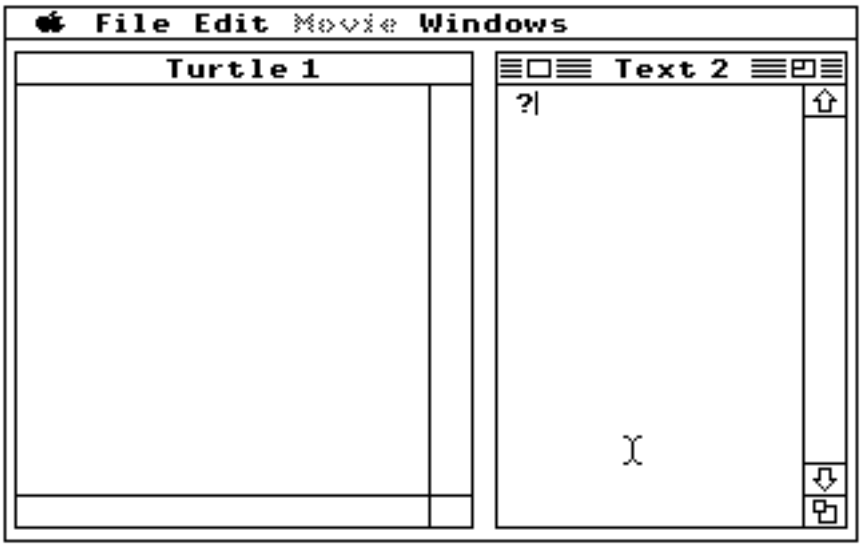
Computer Science LOGO Style: Introduction to Programming
Brian Harvey
The MIT Press
This is the most comprehensive course on Logo programming I am aware of. In fact, it's a three volume set. Unlike many Logo books, this one doesn't concentrate on simple graphics for grade school children. This is a full-blown programming course that will teach you quite a lot about Logo, programming, and computers.

Chapter 2 – Getting Started with 3D Logo

This chapter introduces the basic concepts of the Logo language using example programs. You'll start by creating simple programs that draw lines on the screen, then gradually build on this as you learn about Logo procedures, variables, and lists. Along the way, you'll learn about the 3D Logo environment as you explore the text window, learn to use edit windows, and draw in the turtle window.

Your First Logo Program

How you start 3D Logo depends on how you installed it. If you used the installer to create a set of floppy disks, start by booting OneDiskLogo. From the Finder or any other program launcher, run the program Logo3D.Sys16. Either way, you will end up with a screen that looks like this:



In this manual, we'll assume you already know how to use your computer. You don't have to be the world's greatest expert, but you should already know how to use pull down menus, how to open or close a window, how to edit text in a standard window, and so forth. If you don't, you'll need to read parts of the books that came with your computer to learn these things as you work your way through this chapter.

When most programs start, there is a single window on the screen. That's because most programs do one thing. 3D Logo has two windows. The one on the left is a turtle window; this

is where you'll draw pictures. The window on the right is a text window. The text window is used to type Logo commands.

Tip You might not like the way the windows are arranged, and for some kinds of programming, you might not even want to start with these two windows. You can tell 3D Logo how to arrange the windows, which ones to start with, and what graphics resolution you want to use. To find out how, look at the description of the Startup Options... menu command in Chapter 7.

Drawing Lines with the Turtle

Let's see how this works right away. Type

```
FORWARD 30
```

The triangle is the turtle. It's location shows you where you will start drawing from, and the point of the triangle shows you which direction the turtle will move. The FORWARD command tells the turtle to go forward 30 steps. On the Apple IIGS, each step is one pixel long. As the turtle moves, it draws a line.

Tip In all of the examples in this book, Logo commands are shown using uppercase letters. You don't have to type them that way; we're just using uppercase letters to make it easier to tell when we're talking about a Logo command.

If you've looked ahead at any of the Logo sample programs in this book, they probably look pretty cryptic. That's strange, in a way, because Logo generally uses easy to understand words like FORWARD for the commands. A lot of commands are used so often, though, that typing them out gets to be a drag, not to mention a waste of space and typing time. To save time, the most common Logo commands have two-letter abbreviations. For example, FD is the abbreviation for FORWARD, and

```
FD 30
```

does exactly the same thing as

```
FORWARD 30
```

Clearing the Screen

If you've tried the abbreviation for FORWARD, the turtle may have already marched off of the screen. Any time you want to start over with a fresh screen, type the command

```
CLEARSCREEN
```

This erases everything in the screen and moves the turtle back to its starting position. Most of the time, you'll want to use the abbreviation CS.

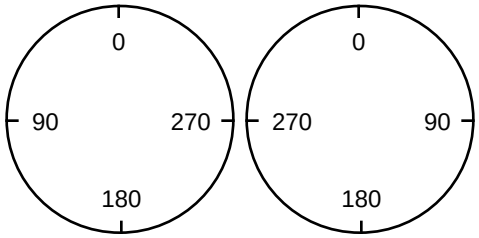
Turning the Turtle

You need to turn the turtle to draw anything interesting. You can turn the turtle left with the LEFT command, or right with the RIGHT command. These commands have abbreviations, too. They are LT for LEFT, and RT for RIGHT.

The turtle turns in degrees, so turning the turtle right 90 degrees is like making a right-hand turn with a car. No matter which way the turtle starts,

```
RIGHT 90
```

makes a right-hand turn. Turning 180 degrees in either direction turns the turtle around.



Logo's REPEAT command lets you tell Logo to do more than one thing at a time. The line

```
REPEAT 4 [FORWARD 20 RIGHT 90]
```

tells Logo to repeat the commands in the brackets four times. Logo knows there are two commands inside the brackets because it knows FORWARD only needs one number for an input, so the word right after 20 has to be a new command.

It turns out that the way you type this line is important. The REPEAT command is a single line, with no RETURN keys anywhere in the line. If the text window isn't wide enough to display a line, the line will get wrapped in the text window, but you can't type a RETURN key to break the line up yourself.

If you haven't already tried the REPEAT command, type the line now. If the turtle isn't on the screen, use CLEARSCREEN first. Logo will draw a small square.

Exploring Black on white is fine, but you can use color, too. Try SETPC followed some number from 0 to 15, then draw a line, like this:

```
SETPC 4
REPEAT 4 [FD 20 RT 90]
```

You'll find more information about color in Chapter 6. You might want to try the SETBG command, too, to change the background color.

Creating Procedures with TO

Logo programs are usually written as a series of very short procedures. There are several ways to create procedures. We'll start with a simple, interactive command called TO.

TO teaches Logo to do something new. The procedure you create will look and work just like the built in commands you've used so far. Of course, you have to give the command a name. The TO command expects you to type the name of the procedure you are creating right after TO.

To see how this works, let's create a procedure to draw a square. Start with the TO command, like this:

```
TO Square
```

So far, Logo always shows a ? character when you can type a command. Once you type TO, though, you start entering a procedure. Logo switches to a > character to remind you that you are typing in a procedure, now. As long as the > prompt is shown, the commands you type aren't executed, like they have been so far. Instead, Logo saves these commands and executes them when you type the name of the procedure.

Our sample procedure will draw a square, so you can type the same line you used a moment ago to draw a square in the turtle window, or you can switch to the abbreviations, like this:

```
REPEAT 4 [FD 20 RT 90]
```

When you finish typing all of the commands for the procedure, finish it off by typing

```
END
```

This tells Logo you are finished entering the procedure. Logo tells you the procedure is defined.

Running your procedure is as easy as using any other Logo command. To draw a square, type

```
Square
```

Logo procedures can call the built-in procedures, or the ones you define. Once you enter a procedure, anyone can use it. Here's two classic Logo procedures that show how this works. Type them in and try both `Flag` and `Flower`.

```
TO Flag
FD 20
SQUARE
PENUP
BACK 20
PENDOWN
END

TO Flower
REPEAT 10 [FLAG LEFT 36]
END
```

There are a few new commands in `Flag`. `PENUP` raises the turtle pen, so it doesn't draw a line as it moves, and `PENDOWN` puts the pen down so it will start drawing lines again. `BACK` is the opposite of `FORWARD`: the turtle moves backward. All of these commands have abbreviations. You can find a list of them in Chapter 8. It's a pretty good idea to start exploring that chapter now, in fact, reading about new commands as we try them, and scanning the chapter for similar commands you might like to try.

Tip If Logo seems a little slow drawing these figures, well, it is. The reason is that it's doing a lot more work than the picture shows. Every time the turtle moves, Logo has to draw six lines – three to erase the old turtle, and three to draw the new one. You can draw pictures a lot faster by hiding the turtle before you start drawing using `HIDETURTLE`, then showing the turtle when you're finished with `SHOWTURTLE`. You can find complete description for these commands, along with two-letter abbreviations, in Chapter 6.

The Logo Workspace

So far, you've created three procedures. Now try this: Close the text window completely. You'll be asked if you want to save the window to disk; say no. Now open a new text window by pulling down the File menu and picking New. Finally, type

```
CS
FLOWER
```

The point of this little exercise is to introduce the concept of the Logo workspace. The text window you use to type command is just a scratch pad. You can use several of them, close them, delete all of the text in the windows, or even select lines from the text window and execute the lines a second time.

When you create a procedure, though, it's independent of the text window. Procedures are stored in Logo's workspace, where they can be used from any text window.

Listing Procedures in the Workspace

Of course, after a while you might loose track of just what is in the workspace.

```
POALL
```

prints everything in the workspace, including variables and property lists – two topics we haven't talked about, yet. You can also print just one procedure with the PO command:

```
PO "Flower
```

These commands show the procedure just like you entered it. You can quickly change the procedure with a few mouse clicks and keystrokes. Let's say you want to create a slightly larger square. Start by typing

```
PO "Square
```

This gives you a listing that looks like this:

```
TO Square
REPEAT 4 [FD 20 RT 90]
END
```

To change the size of the square, change the 20 to a 30. When you are through, select all three lines and press the return key. When you enter a procedure with the same name like this, it replaces the old procedure. This is a quick way to make short changes to a procedure.

Printing the entire workspace prints a lot of stuff, when you may just want to get a quick list of the procedures that are in the workspace. The POTS command lists just the name of the procedures.

Some of these names seem pretty cryptic at first. Think of them as "Print Out" something. In the case of POTS, think "Print Out TitleS."

Exploring There are several varieties of the PO command. You can find a complete listing in the reference section. A few you might want to try right away are POT, which prints a single procedure title, and POPS, which prints just the procedures in

the workspace. True, so far all we know how to put in the workspace is a procedure, but that will change shortly!

Editing Procedures with Edit Windows

Editing a procedure in the middle of the text window is a neat, fast trick, and it works especially well when you've just entered a short procedure and want to try a quick change or two. Sometimes, though, you will want to edit a procedure with a separate window. The slickest way to do that is with the `EDIT` or `EDITS` command; they open up a separate kind of window called an edit window.

To see how edit windows work, edit the `Square` procedure with this command:

```
EDIT "Square
```

The `EDIT` command opens a new window with `Square` in the window. Edit windows are different from text windows: When you type a line in a text window and press RETURN, the command is executed right away. When you type a command in an Edit window and press return, you just change what's in the edit window.

Change the length of the sizes of the square from 30 (or 20, if you didn't make the last change) to 40. Now select the original text window, either by dragging the edit window to the side and clicking on the text window, or my personal favorite, by pulling down the Windows menu and selecting the name of the text window. When you switch back to the text window, you'll see a new line, "Square Created." Any time an edit window is the front window and you close it or select a different window, Logo enters everything in the text window into the workspace.

Tip What if you have made several changes in an edit window and don't *want* to enter them in the workspace? In that case, delete everything in the edit window and *then* close it.

In our sample procedures, you actually have two procedures, `Square` and `Flag`, which are closely related. When you change the size of one, you probably want to change the size of the other. `EDIT` can edit both procedures at once, in the same edit window, by putting the names of the procedures in a **list**. Lists are something we'll talk about more later. So far, you've used **words** to name a procedure. Words are the equivalent of strings in other languages, but they look a little strange, since there is no closing quote mark. In a list, you type the names without quote marks, and put all of the names inside brackets, like this:

```
EDIT [Square Flag]
```

`EDITS` doesn't need any names. It's the plural form of `EDIT`. The `EDITS` command opens an edit window that shows everything in the workspace.

Exploring Actually, you can hide procedures from EDITS, and from all of the other commands that work on "everything" in the

workspace. Logo has a series of bury commands that hide stuff from normal view, and some unbury commands that retrieve the things you have buried. Try BURY and UNBURY; you'll find descriptions and examples in the reference chapter.

Removing Procedures from the Workspace

There comes a time when you might want to clean up your workspace. After all, the procedures do take up space. Then again, you might have tried a few things that didn't work out like you wanted, and you might want to get rid of the failures so you don't see them every time you list all of the procedures in your workspace.

ERASE erases a procedure from the workspace. You can erase a single procedure or a list of procedures. Here's two samples you can try; they will erase the three procedures you've created so far:

```
ERASE "Square
ERASE [Flower Flag]
```

Exploring There are several other erase commands that work just a bit differently. They are all grouped together in the reference chapter. Two you might find interesting right away are ERALL and ERPS.

Variables

Creating Variables with MAKE

Logo's MAKE command is used both to create a new variable and to assign a value to a variable. To create a variable called size and save, say, the size of a square, you could use

```
MAKE "size 20
```

The first thing MAKE looks for is the name of a variable. With the single exception of the TO command, anytime you want to give a name to Logo, you use a single quote before the name. Right after the name is the value you want to stuff into the variable.

When you want to use the value stuffed inside of a variable, use a colon before the name. Here's Square, reworked so it uses size to figure out how big the square should be.

```
TO Square
REPEAT 4 [FD :size RT 90]
END
```

You can put this variable to use right away to create a cascade of ever larger squares.

```
MAKE "size 10
REPEAT 8 [Square MAKE "size :size + 10]
```

Quotes and Colons and Nothing, Oh My!

Take a close look at that last MAKE command. It's still setting the variable size; you can see the name with the quote mark right after the name of the MAKE command. The next part is :size + 10, which MAKE treats as a single lump, adding 10 to whatever used to be in the variable size.

Using quotes in one place and colons in another probably seems awkward, and it is, at first. The way to keep the two punctuation marks straight is to remember that "size is the *name* of the variable, while :size is the *value* of the variable. Size with nothing else is a *procedure*; Logo will look for a procedure called size, run it if one is found, and use the value size returns.

This triple use of a single name is both the bane of people who are just starting to use Logo, and one of the most powerful features of the language. After all, if there wasn't some reason for all these punctuation marks, Logo would have just left them out, like most other languages. The reason is tied to the fact that Logo programs can change themselves as they run, adding new procedures or using variables to do tricks that would be very hard to duplicate in a traditional programming language. For example, you could us a variable value for the first parameter to MAKE, like this:

```
MAKE :variable 20
```

This single command can initialize any number of different variables. What Logo actually does is look inside the variable name for the value, which in this case is the name of the variable to set!

If all of this seems a little strange at first, don't feel left out. It seems a little odd to most people the first time they see it, especially to people who are used to other computer languages, like BASIC or C. The important thing, for now, is just to remember that *quotes mean names*, and *colons mean values*. As you learn more about Logo and start to explore some of its power, you'll gradually pick up a lot of tricks that make this oddity one of the most powerful features of the Logo language.

Printing a Variable

There are several printing commands that let you look at the value of a variable. One of them is SHOW. Here's how you can look at the value of size using the SHOW command:

```
SHOW :size
```

Just for the sake of experimenting, try the same command with a quote mark:

```
SHOW "size
```

This time, SHOW shows the name of the variable, which is, after all, what you asked it to do.

☕ **Exploring** There are several ways to print a value, and each has it's own unique features. Two other printing commands you may want to use from time to time are PRINT and TYPE. ☕

Logo Numbers Come in Two Flavors

Like most computer languages, Logo uses two different kinds of numbers. Whole numbers, like 4, -16 and 10000, are called integers. Numbers with a fraction part, like 4.27 or 3.14159, are called floating-point numbers. You can use either kind of number in Logo, and you can mix them any way you like.

Word Variables

You can save words in a number, too. Logo words are the equivalent of strings in many other computer languages, but they work a little differently. There is no closing quote on a Logo word. A word ends when Logo finds the first punctuation mark, usually a space. To imbed a punctuation mark in a word, use a \ character right before the punctuation mark. Here's an example that stuffs a complete word into a variable, then prints the value.

```
MAKE "Hi "Hello,\ world.
SHOW :Hi
```

Besides spaces, all of these characters will stop a word:

```
[ ] ( ) = < > + - *
```

You have to put the \ character before any of these characters to stuff them into a word. To put a \ character in a string, use two in a row, like this:

```
MAKE "Word "Here's\ how\ you\ put\ in\ a\ backslash:\ \\
```

Lists

The last kind of Logo value that you can stuff into a variable is the list. Even if you're a very experienced programmer, you may not have used a language that handles lists. Lists are enormously important in artificial intelligence languages like LISP (LISt Processing) and Logo, but are completely missing from language like C and Pascal.

A list is just a series of values. You put all of the values in square brackets to form the list. About the only difference between stuffing values into a list and putting a single value in a variable is that you don't need a quote before words.

```
MAKE "Hi [Here's an example of a list of words.]
PRINT :Hi
```

You can put any kind of variable you want in a list, including another list. You can even mix different kinds of variables in a list.

```
MAKE "Sample [1 3.14159 [Lists in a list] word]
```

Lists are very important in Logo, but unless you've used LISP, you've probably never seen this sort of data before. The best way to learn about lists is to use them in a variety of ways. You'll see lists used quite a few times in several different and very powerful ways throughout the rest of the introductory chapters.

Exploring Manipulating lists – pulling them apart, putting them together, searching a list, and so forth – is what makes Logo a powerful artificial intelligence language, not just a kid's graphics language. In fact, our Logo was tested in part by working problems from a LISP artificial intelligence textbook!

You'll see examples of lists and commands that manipulate lists and words scattered throughout this book. It's worth looking over the various commands that manipulate lists now, though, too get a preview of some of the commands you can use. Check out "Words and Lists" in Chapter 8 for other list manipulation commands, plus a lot of examples.

Logo Variables Do It All

So far, you've stuffed integers, floating-point numbers, words and lists into variables. How does Logo know what a variable is supposed to hold? After all, anyone who has used BASIC or

Pascal or C knows that stuffing the wrong kind of value into a variable either won't work at all, or can cause serious problems.

Well, the answer is that Logo doesn't really care. Logo variables are general purpose buckets. You can stuff whatever you want into a variable. In fact, a variable can hold a number on one line, a word on the next, and a list on yet another line, and it will work just fine!

```
MAKE "Var 14
PRINT :Var
MAKE "Var "XIV
PRINT :Var
MAKE "Var [1 2 3]
PRINT :Var
```

Logo even takes this mix-and-match approach to variables one step further: technically, there really are only two kinds of variables in Logo. Logo uses lists and words, and that's it. Numbers are just a special case of a word, with some special properties. Later, you'll find some Logo commands that work on words, and every one of them will work on numbers, too!

Let's look at an example to see how this happens. Try this:

```
MAKE "a 01
MAKE "b "02
```

The first line creates a variable called A and stuffs the integer 1 into the variable. Next, the word "2 is stuffed in the variable B. This is important: B really does have a word, not an integer. To see what the difference is, try these print commands:

```
PRINT :a
PRINT :b
```

The value of 01 is 1, so Logo prints 1. The value of "02, though, is a word, not a number, and you get both characters.

Since a number is just a special case of a word, though, you can use words in mathematical expressions, as long as the word translates into a number. This really will work, and will print 3:

```
PRINT :a + :b
```

So words can be used as numbers. The opposite is also true. `FIRST` is a command that returns the first character from a word (or the first item from a list).

```
PRINT FIRST "ABC
```

will print A. But numbers are just a special case of words, and

```
PRINT FIRST 3.14159
```

prints 3!

Of course, numbers really are stored a different way than words. If they weren't Logo programs would be very, very slow. There are a few places where the difference is important, like when a number has extra digits.

```
PRINT FIRST 001
```

will print 1, not 0.

Exploring There are times when you need to know just what a variable contains before you try to do something with it. Is a variable a number? Is it a list? Since variables can hold anything, you need some way to find out. You'll find a series of predicate commands – commands that end in a P – that can look inside a variable to see just what's there. They are `LISTP`, `WORDP`, `FLOATP` and `INTEGERP`. You can find examples in Chapter 8.

What About Arrays?

If you've used another computer language before, like BASIC or Pascal, you've probably used arrays to hold a fixed number of values. Maybe you used an array to work with a matrix for a math class, or to hold a series of names for a simple database.

Well, Logo doesn't have arrays. In fact, it doesn't need them. Lists do the job quite well. Just because you can mix the kinds of values you stuff in a list doesn't mean you have to mix them!

In Logo, an array is a list. An array of arrays is a list of lists.

Variables as Parameters

Traditionally, Logo programs are written as a series of short procedures that are debugged as they are developed. In these short procedures, the most important kind of variable isn't usually the ones that are create with the `MAKE` command. The most commonly use variables are actually parameters to procedures.

There are several differences between parameters and the variables you have created with `MAKE`. The most obvious is the way the variable is created in the first place. To create a parameter, you put the variable name on the `TO` line you use to create the procedure, like this:

```
TO Square :length
```

Once you've created a parameter, you can use the value or even change the value just like you did with variables created with MAKE. Here's a complete version of Flag and Square that use parameters to set the length of the lines:

```
TO Square :length
REPEAT 4 [FD :length RT 90]
END

TO Flag :length
FD :length
Square :length
PU
BK :length
PD
END
```

You can see how to pass a parameter by looking at how Flag calls Square. Here's another example, this time drawing a flag:

```
Flag 25
```

The second big difference between parameters and variables created with MAKE are where they can be used. In very technical books you'll see this called the **scope** of the variable. If you've been following along for the last few pages, you've created several variables with MAKE, like HI, A, and B. These variables are global; they're stuffed into the workspace just like procedures. There's even a set of commands to look at the values and erase them, just like for procedures. Try that now:

```
PONS
```

You can think of PONS as Print Out NameS. It shows all of the variables in the workspace, along with their values. One of the things you didn't see, though, was a variable called length. That's because parameters are created when the procedure starts to run, and go away as soon as the procedure is finished. They can only be used and set from inside the procedure, too. Flag and Square both have parameters called length, but these are two separate variables. For example, you can't use this shortcut:

This Does Not Work!

```
TO Square
REPEAT 4 [FD :length RT 90]
END
```

```
TO Flag :length
FD :length
Square
PU
BK :length
PD
END
```

The reason this doesn't work is that `length` is only available inside of `Flag`. Even though `Square` is called from `Flag`, it can't use `Flag`'s variables.

Exploring When you create a variable inside of a procedure with `MAKE`, the variable is still entered into the workspace. You can create something called a local variable, though, using `LOCAL`. Local variables exist only inside of the procedure, just like a parameter.

Math With Logo

You've already seen some simple math operations in Logo. In the next couple of paragraphs, we'll take a quick look at how Logo handles math operations. There aren't many surprises here, so this section will be fairly quick. If you are completely new to computer languages, this may go a little to quickly. If that's the case for you, there are several things you can do to find out more about how Logo handles math operations:

- Chapter 8 of this manual has a more in-depth description of Logo's math capabilities. Look in the section "Numbers and Arithmetic."
- Beginner's books on Logo all deal with math operations on one level or another. Look under "Logo" in the subject listing of Books in Print at any large bookstore for a long list of Logo books you can choose from.
- Probably the best idea is just to try some things as you read this section. The worst thing that will happen if you make a mistake is Logo will tell you what the mistake was – all in all, not a bad way to learn!

Addition, Subtraction, Multiplication and Division

Any Logo command that will take a number will also take a mathematical expression. For example, `FORWARD` will take a constant, like

```
FORWARD 40
```

or an expression, like

```
FORWARD 20 + 20
```

Anywhere you can use a number, a variable will do, too, as long as the variable contains a number. You'll see expressions like this a lot:

```
MAKE "Distance 20
FORWARD :Distance + 20
```

Logo supports the four standard math operations you see on just about any calculator. The operations are:

`a + b`	Add `a` and `b` together.
`a - b`	Subtract `b` from `a`.
`a * b`	Multiply `a` and `b`.
`a / b`	Divide `a` by `b`.

▲ **Warning** Unlike the other math operators, the `/` character isn't a separator. That has a very important consequence: When you type something like `:A+:B`, Logo knows you are adding the value of :A and :B together, since `+` is a separator. When you type `:A/:B`, though, Logo treats the `/` character as a part of the name of the variable.

The proper way to type an expression that uses a division operator is to put spaces on both sides of the `/` character. ▲

Of course, you can mix these operations, putting more than one in a single expression. The only thing you have to be careful of is what order the operations are done in. For example, what happens when you type

```
FORWARD 1 + 2 * 3
```

There are two ways to handle the math in this case. Both make sense, and both are used in various computer languages. The way most algebra students learn to handle this expression is to do the multiplication first, so the answer is 7. It's also possible to do the operations left to right, which would move the turtle forward 9. Logo uses the same conventions as algebra, so this example would actually move the turtle forward 7 units.

So what if you want to change the order of operations? Logo uses parenthesis, just like you use in algebra. The only difference is that in algebra, you can leave out a multiplication sign just before or after a parenthesis, so 3(1+2) is pretty common. Logo makes you put the multiplication sign in.

Here's our example, using parenthesis to change the order so the turtle moves forward 9 units:

```
FORWARD ( 1 + 2 ) * 3
```

Negative Numbers

Numbers can be positive or negative, of course. It's perfectly natural, and works just fine, to move backward this way:

```
FORWARD -10
```

You can also put a minus sign in front of a variable, like this:

```
FORWARD -:distance
```

Either way, the effect is the same as subtracting the number from zero. There is a problem with the unary subtraction operator, as this operation is known. To see what the problem is, let's have Logo echo a simple list of two numbers back to us. When you type

```
PRINT [1 2]
```

Logo echoes back with

```
1
2
```

That's natural enough. Now try

```
PRINT [1-2]
```

Be sure you type the line with no spaces on either side of the - character. Logo decides this is a subtraction operation, and says

```
-1
```

Now try

```
PRINT [1 -2]
```

making sure you put a space before the - character. This time Logo treats the - operator as a minus sign, and responds with

```
1
-2
```

This isn't what you would expect if you've used other computer languages. Logo is a list processing language, though, and it really did need some rule to tell the difference between subtraction and negating a number. The actual rules used are a little complicated, but you're safe

with a simple rule of thumb: Always put a space before a minus sign, but never put one before a subtraction operation. The complete set of rules is in the section "Expressions" in Chapter 6.

Functions

The last thing you can put in a Logo procedure is a function. A function is a procedure that returns a result. For mathematical expressions, of course, the result has to be a number.

There are a lot of built-in Logo functions. One example is SQRT, which takes the square root of a number. Here's a simple example that uses the SQRT function to find the length of the hypotenuse of a right triangle:

```
PRINT SQRT :A * :A + :B * :B
```

Exploring The section "Expressions" in Chapter 8 lists all of the functions that are built into Logo. If you want to write your own Logo functions, check out the OUTPUT command.

Reals and Integers

Back in the section "Logo Variables Do It All," you learned that there are really two kinds of numbers in Logo. The first kind are whole numbers, which are called integers in computer circles. The other kind are real numbers, which can have a fractional part.

So which do you use?

The answer is pretty neat: Don't worry about it. Logo figures things out for itself, and uses whichever kind of number makes sense. In those rare cases when some Logo function needs a real number, and you give it an integer, Logo just converts the number. If a function needs an integer, and you supply a real number, Logo rounds the real number to the closest integer.

There are only three places where you need to be aware of what sort of number you are using:

- When you print a number, you may want a little control over the format.
- If you are doing operations on very large integers, you might end up with an integer that is too large for Logo to handle. You can avoid a math error by converting the integers to real numbers. (The largest integer 3D Logo can handle is 2147483648.)
- When you are doing complicated drawings, Logo can turn the turtle faster if you use integers for angles instead of real numbers.

In all three cases, you can use Logo's INT and FLOAT functions to convert back and forth from integer to real numbers.

Loading and Saving Logo Programs

Once you create a set of procedures or a program, you'll want to save them to disk so you can load the procedures again. The simplest way to save a procedure is to use Logo's SAVE command. You give the name of the file as a word. To save your workspace to a file called MYFILE, you would use the command

```
SAVE "MYFILE
```

This saves all of the procedures, variables and property lists – everything in the workspace – to the disk. Even after you quit the program, you can reload everything that was saved and put it back in the workspace with the load command, like this:

```
LOAD "MYFILE
```

Note Technically, SAVE doesn't save everything in the workspace. There are some advanced Logo commands that can bury (or hide) procedures, variables and property lists. Anything that is buried won't be saved. Unless you're using these advanced commands, though, SAVE saves everything in the workspace.

There is one thing you have to be aware of when you use LOAD and SAVE, and that is how you go about picking the disk and folder where the file will be saved. There are several ways to do this. Which one you pick will depend on the situation and how familiar you are with path names.

The simplest way to pick a location for a file is to start with the ONLINE command, which lists all of the disks that you have in your computer. Once you have a list of the disks, you change the current default location to the correct disk with the SETPREFIX command. If you want to change the prefix to a disk named MYDISK, use the command

```
SETPREFIX ":MYDISK
```

When you set the prefix to a disk, you're located in the disk's root directory. The CATALOG command shows all of the files that are on the root directory. One of the things the CATALOG command shows is the type of each file. Some of the files may have a file type of DIR. Files with a file type of DIR are folders. You can set the prefix to a folder with the SETPREFIX command, too. If you want to set the prefix to a folder called MYFOLDER, use the command

```
SETPREFIX "MYFOLDER
```

There's an important difference between this command and the one we used just a moment ago to set the prefix to a disk. The first command used a disk name, which starts with a colon character. It's this leading colon character that tells Logo you want to set a prefix to a disk, not a folder. In the second case, the folder name doesn't start with a colon.

You can keep right on using the SETPREFIX command to move deeper and deeper into folders. You can even use the CREATEFOLDER command to make a new folder. Once you get to the proper folder, use the LOAD or SAVE commands with the name of the file you want to load or save.

☕ **Exploring** You can use several shortcuts with file names. For example, it's possible to combine disk names, folder names and file names to form a full path name, so you don't have to set the prefix before using the LOAD or SAVE command. For more information about file names, look in the section "File Names" in Chapter 8.

There is another way to save your workspace, too, and you even get a chance to edit it first. Try using the EDITS command to open the workspace, then make any changes you want and save this file. You can load the file you save with the LOAD command. You can also open the file. Opening the file doesn't enter the procedures in the workspace. You can enter any of them, or even all of them, in the workspace by selecting the procedures and pressing RETURN. ☕

Chapter 3 – 3D Pictures with 3D Logo

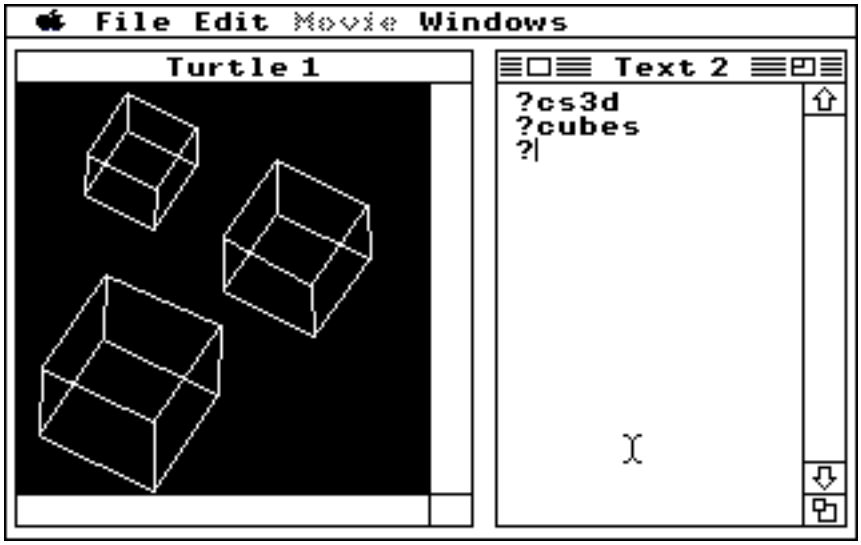
The Cubes Program

This chapter shows you how to create your own 3D pictures with 3D Logo. Before we get too far into the chapter, though, let's take a look at an example. Start by opening Cubes. Select the entire screen by pulling down the Edit menu and picking Select All, then press the RETURN key. Finally, close the Cubes window.

It's time to put on the 3D glasses! Make sure the red lens is on the left. It also helps to dim any bright lights, although normal room lighting won't destroy the effect. Now type

```
CS3D
Cubes
```

You'll see three cubes on your screen, like this, although yours will be jumping out of the screen instead of laying flat on the page.



Problems? If you had trouble locating the Cubes file, use the Open command to look around on the various disks you have. On a hard disk, the file is inside of the Samples folder. If you're using a single floppy disk, look on the samples disk.

There are three common reasons why the picture might not look three-dimensional:

- You must use a color monitor. The 3D turtle has a lot of uses besides drawing true 3D pictures, but to see the 3D pictures, you must use a color monitor.
- Make sure you typed `CS3D` to turn on the 3D turtle. If you did this, without the glasses you should see a red and blue drawing on a black background.
- If the drawing just doesn't look three-dimensional – if it looks like red and blue lines, for example – give it some time. Try playing with the room lighting, too. We've found that, roughly, the older the person, the longer it takes to relax and see the 3D picture. Eventually, you should see gray or purple lines on a black background. ◭

How the 3D Display Works

You may already see how the 3D display works, but let's stop and talk about it for a moment. Some of the ideas will help you create realistic 3D pictures, even if you already know the basic concepts.

Hold your finger up at arm's length in front of a background that is at least a few feet away. Cover your left eye and look at your finger, but notice where it is against the background. Now, without moving your finger, uncover your left eye and cover the right one. Your finger will jump to the left against the background.

Your brain is a powerful image processor. It's so powerful, in fact, that you don't notice how you see three-dimensions unless you really stop to think about what is happening and use tricks like covering one eye or the other. Basically, though, you are seeing two pictures at a time, each from a slightly different vantage point. Your brain automatically combines these images to tell you how far away things are.

Of course, there are limits. If something is so far away that it doesn't shift against the background, you can't really tell anything except that it's not close. People with one eye, or with severely limited vision in one eye, can't tell how far away things are – they have no depth perception.

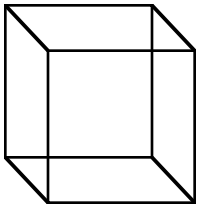
Logo is using a pair of special glasses to give your brain two pictures of the same scene from slightly different viewpoints. Since you only have one computer screen, there's obviously a trick involved. Logo draws one picture using shades of red and one using shades of blue. The red picture comes through the red lens just fine, but is blocked by the blue lens, so your brain sees this as the left-eye view. The blue picture is the right-eye view.

Note 3D Logo is using color to split the images between your left and right eye, which means you can only draw black-and-white

pictures in three dimensions with 3D Logo. There are other ways to do the same trick, of course. At some theme parks, they use polarizing lenses and two projectors to get the same effect, but because they are using polarizers, the theme parks can still show color. So why doesn't 3D Logo do this? Basically because you can't display two polarize images on a standard computer monitor, and we didn't think you'd want to spend tens of thousands of dollars for a special monitor!

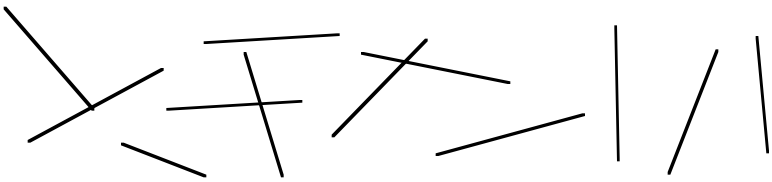
Realistic 3D Pictures

It may take a little practice, but the cubes demonstration should look pretty realistic. In fact, some of our younger testers even reach out to grab the cubes! Some of the simple lines you'll be drawing as you explore the 3D turtle won't look like they are really three dimensional, though. The reason is simple. Your brain is so used to processing three dimensional, solid images like the ones you see in the real word every day that it desperately wants to see the cube as a three dimensional object. You can even see a cube with a simple line drawing, like this one:



You can't tell which is the front and which is the back, and in fact, many people can mentally flip the front and back of the cube as they look at it. With the cube, then, your brain really wants to see three dimensions, and Logo just gives it a helpful nudge in the right direction.

Now look at this set of lines – the same number as in the cube:



This time, there is no order, and your brain isn't picking out an image. This picture doesn't look three dimensional at all. If you draw single lines which aren't connected to anything in 3D Logo, your brain has a hard time using the depth information in the picture. You may see a jumble of red and blue lines instead of a three dimensional set of lines. You might even get a little disoriented, since your brain doesn't understand why some of the lines appear in one eye and not the other.

The secret to drawing realistic pictures with 3D Logo is really very simple: Try to help the brain by giving it a picture of solid objects or lines that connect. Don't try to fool the brain by drawing points or lines that your brain rarely, if ever, sees floating in air in real life.

The 3D Turtle

Creating 3D pictures with 3D Logo is really very simple. The first step is to clear the screen and start out in the 3D viewing mode. You can do both with CLEARSCREEN3D. CLEARSCREEN3D works pretty much like the CLEARSCREEN command you learned in Chapter 2, but in sets up a 3D screen instead of a 2D screen. The turtle moves to the center of the screen, pointed up. You've already used the abbreviation for this command, which is CS3D.

The other way to switch to a 3D turtle is with SHOWTURTLE3D, abbreviated as ST3D. You can switch back to a 2D turtle with SHOWTURTLE. Normally this isn't very important, although you might want to do some tricks by mixing 2D and 2D drawings. It's still an important point, though, since you might be in the habit of using HIDETURTLE and SHOWTURTLE to speed up your drawings. When you're using the 3D turtle, you need to use SHOWTURTLE3D instead of SHOWTURTLE.

Rotating Out of the Screen

You can turn the 3D turtle left and right with the LEFT and RIGHT commands, just like you did with the 2D turtle. In fact, everything you did with the 2D turtle will still work with the 3D turtle. You can even set the pen color – but the result with the 3D turtle is black and three shades of gray, not the 16 colors you had with the 2D turtle. As long as you only use 2D turtle commands, you're locked in the screen, and all of the lines will be gray (or purple, without the glasses).

Two new rotate commands free your pedestrian turtle, so it can take to the sky. ROTATEOUT (abbreviated RO) turns the turtle up, rotating it out of the screen. ROTATEIN (abbreviated RI) turns the turtle down. You can put these to work right away, drawing a line that moves out of the screen. Of course, like we mentioned in the last section, a single line doesn't look very three dimensional, but if you look at it carefully enough, you should see the line coming out of the screen.

```
RO 45
FD 50
```

Rolling the Turtle

Imagine the turtle as a bird, or a plane – or maybe just a really strange turtle with wings. Flying around in the air, you can point the turtle in any direction with the four turning commands you already know. There is one more way to rotate the turtle, though, and that's to roll the turtle.

Rolling the turtle doesn't change the direction it's pointed. Rolling the turtle is like a bird or plane dipping one wing while lowering the other, spiraling so the nose stays pointed straight ahead. ROLLRIGHT (abbreviated RLR) rolls the turtle so it's right edge goes down and the left edge rolls up, turning in a clockwise direction compared to the way the turtle is pointed. ROLLLEFT (abbreviated RLL) rolls the turtle the other direction.

Some 3D Shapes to Play With

To see how these commands work, let's dissect the Cube procedure from the Cubes sample you ran at the start of the chapter.

The Cube procedure has a single parameter, the size of the cube. It starts like this:

```
TO Cube :size
```

A cube is basically a 3D version of a square, so that's how we'll draw it. The first step is to draw a square with posts pointed up at each corner. To draw a square, you use a command like this:

```
REPEAT 4 [FD :size RT 90]
```

To put a post pointed straight up from where the turtle is, you rotate out 90 degrees, draw the line, back up to the starting place, and rotate in 90 degrees so the turtle ends up right back where it started. The Logo command to draw a post are

```
RO 90
FD :size
PU
BK :size
PD
RI 90
```

Putting these two ideas together, here's the line that draws the bottom of the cube and the four corner posts:

```
REPEAT 4 [FD :size RO 90 FD :size PU BK :size PD RI 90 RT 90]
```

The next step is to move to the top of the starting post. It works pretty much the same way as drawing a post, but you don't draw the line or back up after you get to the top.

```
PU
RO 90
FD :size
RI 90
PD
```

Next, you draw a square to form the top of the cube.

```
REPEAT 4 [FD :size RT 90]
```

The last step is to move the turtle back to it's starting point. This step doesn't draw anything, but it makes it easier to use the Cube procedure to draw more complicated pictures. By putting the turtle back where we found it, we don't have to worry about what the procedure might do to the position or orientation of the turtle when we use it to draw a cube.

```
PU
RO 90
BK :size
RI 90
PD
END
```

With a little work, you can see how to create the first useful addition to Cube. By taking three inputs instead of one, and using them for the depth, width, and height of the cube, you can draw a box – and with a little imagination, that box could be a building in a city, the body of a simple car, or the top of a table in a room.

```
To Post :height
RO 90
FD :height
PU
BK :height
PD
END

TO Box :width :depth :height
REPEAT 2 [FD :depth Post :height RT 90 FD :width Post :height
RT 90]
PU
RO 90
FD :height
RI 90
PD
REPEAT 2 [FD :depth RT 90 FD :width RT 90]
PU
RO 90
BK :height
RI 90
PD
END
```

This one may stretch your imagination a bit. Octahedron draws an eight sided figure in space, where each of the eight sides are a triangle with all of the angles 60 degrees. You could draw it as a series of triangles – but this way is a little faster. This procedure draws the octahedron as three squares connected at the corners. It's a fun shape to play with in three dimensions.

```
TO Octahedron :size
RT 45
Top :size
LT 45
FD :size
RT 90 + 45
Top :size
LT 45
REPEAT 3 [FD :size RT 90]
END

TO Top :size
RO 45
REPEAT 4 [FD :size RI 90]
RI 45
END
```

3D Without the Glasses (The Chemistry Program)

If you think 3D shapes are just a gimmick or a kid's toy, this section should change your mind. There are a lot of very practical uses for the 3D turtle, and this section will show you one of them.

Tip

This is a pretty complicated example. If you've programmed in other computer languages, you'll probably work right through this section without a hitch. If you have never programmed before, though, or if you were never very comfortable programming in other languages, this example may be a bit deep.

If you decide this example is a little beyond where you're at, there are two ways you can approach this section.

First, you can slow down, take the section very slowly, and try a lot of things as you go along. Read the reference section on each new command carefully. Read the introductory sections in the reference manual about lists. Try a lot of

things as you see new ideas and commands. You might spend several hours, or even a weekend or two, just on this example, but you'll learn a lot about programming in general and Logo in particular.

The other thing to do is just read through the section, typing in the commands as you are told to. You'll still see some neat demonstrations, and learn a few things along the way. Later, when you're more experienced, you can come back to this section and try again. ◭

Trying the Chemistry Program

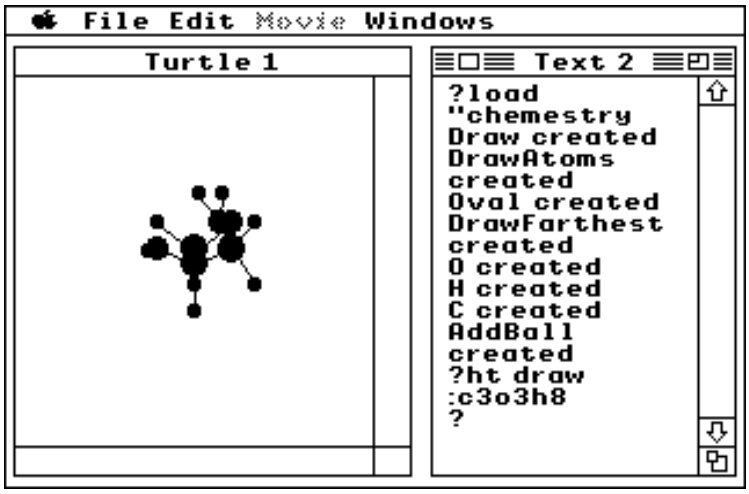
If you've been following along, close all of the windows on the desktop and open a new turtle window and a new edit window. Next, load Chemistry from the samples disk and type these commands:

```
HIDETURTLE
DRAW :C3O3H8
```

△ **Important** The name of the variable is the letter C, the number 3, the letter O, the number 3, the letter H and the number 8. You'll get an error if you use a zero instead of the letter O. △

What you see is a true 3D rendering of $C_3O_3H_8$. In plain English, it's a particle of a liquid chemists call glycerol.

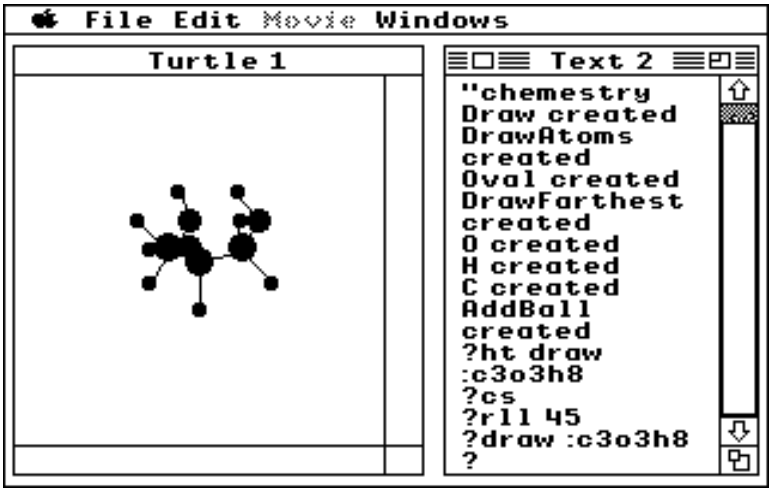
When you draw this on your own computer, one of the things you'll notice right away is that the picture is in color. It's not using the 3D glasses, but the program that draws the molecule does use the 3D turtle. In fact, without the 3D turtle, it would be a *lot* harder to draw this picture.



When you drew cubes and octahedrons in 3D, changing the direction of the turtle rotated the object the turtle drew. The same thing happens with this demonstration. If you want to get a better view of the overlapping oxygen atoms (the red ones), just rotate the turtle and try again. After these commands:

```
CLEARSCREEN
ROLLLEFT 45
Draw :C3O3H8
```

you get the same picture rotated to a slightly different viewpoint, like this:



Exploring There are three other molecules that are already in the Chemistry program. You can draw all of them the same way you drew $C_3O_3H_8$. The molecules are H2O (water), CH4 (methane) and C2H5OH (ethyl alcohol).

How the Program Works

While we won't go through this program line by line, there are some interesting things you can learn about Logo and lists from the program, so we'll look at a few procedures.

Logo uses lists a lot. Because of this, one of the things Logo can do easier than a lot of languages is to treat the contents of a variable as a program. That's an important part of creating this molecule with a short Logo program.

The chemistry program actually uses a separate procedure for each atom. For example, an oxygen atom is a list with O, for oxygen, as the first thing in the list, and two other lists for the two other atoms. The reason there are two connections is that oxygen can form a chemical bond to two other atoms. Hydrogen is an H with a single list, since hydrogen only forms one chemical bond. Starting with the oxygen atom, water (H_2O) looks like this in Logo:

```
[O [H []] [H []]]
```

Each of the hydrogen atoms has an empty list. That's because the atom the hydrogen bonds to has already been taken care of, and we don't need to draw it twice. If we started with one of the hydrogen atoms, instead of the oxygen atoms, we would write

```
[H [O [H []] []]
```

This time, oxygen uses one empty list, since it's already connected to the hydrogen.

Exploring With this in mind, entering new molecules is easy. The chemistry program only has carbon (C), oxygen (O) and hydrogen (H), and it only handles single bonds, but there are a lot of molecules that will work even with just these three atoms. To draw a new molecule, just call `Draw` with a list representing the molecule you want to draw. You can find some sample molecules in a chemistry book.

The procedure that eventually gets called to draw an oxygen atom takes two parameters, one for each atom that attaches to the oxygen atom.

```
TO O :b1 :b2
```

Next, the procedure draws the atom connected by the first chemical bond. Here's the whole line that does the work:

```
IF NOT EMPTYP :b1 [SETPC 1 BK 20 RUN :b1 PU FD 20 PD]
```

The first part of this line is checking to see if the parameter is the empty list. The `IF` statement evaluates a condition. In this case the condition is `NOT EMPTYP :b1`. `EMPTYP` is a procedure that checks to see if a value is an empty list. In this case, `EMPTYP` is checking to see if the parameter `:b1` is an empty list. If it is, then `NOT EMPTYP :b1` will be false, and the `IF` statement will skip ahead to the next line in the procedure. If the parameter isn't the empty list, `NOT EMPTYP :b1` will be true, and Logo will do the stuff in the brackets.

Inside the brackets, most of the commands should look pretty familiar. Taking out one of the commands for a while, you get

```
SETPC 1 BK 20 PU FD 20 PD
```

These commands just draw the gray line that connects the atoms.

The really neat part of the line is `RUN :b1`. This runs the contents of the list you pass the oxygen atom as if it were a program instead of a list in a variable. In our water molecule example, what it runs is

```
H []
```

This calls another procedure, `H`, to draw the hydrogen atom. It could just as easily call some other atom which was hooked to still other atoms. You could even write a program that creates the molecules on the fly, then draw the molecule by running the list you build in the program!

The next line of the oxygen procedure draws the second atom, rotating 107° first, since atoms connected to an oxygen molecule are separated by about that angle.

```
IF NOT EMPTYP :b2 [RT 107 SETPC 1 FD 20 RUN :b2 PU BK 20 PD LT
107]
```

Finally, just before it leaves, the procedure calls another procedure to draw a ball with a radius of 5 pixels (9 pixels across) and a color of 7.

```
AddBall 7 5
END
```

There are some tricks in the program. The biggest problem in drawing the molecule isn't handling all of the rotations to connect the atoms. In fact, in Logo, the only real problem is making sure the atoms that are closest are drawn last, so they are on top of the ones that are farther away. The chemistry program handles this problem by building a list of the atoms as the original molecule is traced. Each time `AddBall` is called, the position, color and size of the ball is stored in the list. Once the lines connecting the atoms are drawn, and all of the atoms in the molecules have been traced, the program goes back and looks for the atom that is farthest away, and draws it

first. It then moves up to the next farthest away, drawing that atoms, and so on until all of the atoms are drawn. If you would like to explore this part of the program, look at `DrawAtoms` and `DrawFarthest` in the chemistry program.

☕ **Exploring** The chemistry program is pretty simple, but it's also easy to expand.

For example, adding new atoms is easy. Chemically, sulfur (S) works pretty much like oxygen. To add sulfur to the program's bag of tricks, start with oxygen, but change the name to S and make the atom yellow.

The biggest limitation of the chemistry program for real molecules is that it doesn't handle double bonds. A double bond happens when, for example, an oxygen atom hooks up with another oxygen to form a molecule, and both of the chemical bonds are used. You can handle this easily with a separate procedure, perhaps called O" for oxygen with a double bond. In this case, the procedure would only need one parameter, like hydrogen, since it will only connect with one other atom. ☕

Chapter 4 – Making Movies with 3D Logo

Making Movies is Easy

Making movies is one of the easiest and most practical things you can do with 3D Logo. From classroom demonstrations to visualizing an object to grabbing bragging rights at the local computer user's group, making movies is a great way to go.

Creating Your First Movie

We're going to learn about movies in 3D Logo the easy way: by making one! Start Logo in the normal way, or if you're already there, clean out the workspace with

```
ERALL
```

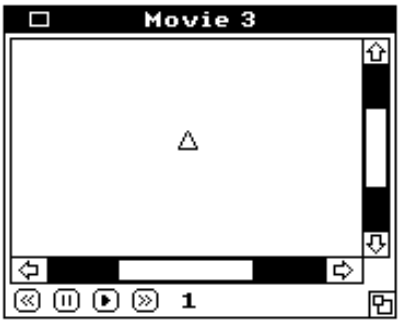
Close any windows except the turtle window and the text window, and enter the Flag procedure from Chapter 2. Actually, this one is a little simpler.

```
TO Flag :s
FD :s
REPEAT 4 [FD :s RT 90]
BK :s
END
```

Try the flag routine to make sure it works:

```
FLAG 20
```

Now close the turtle window and open a movie window. To open a movie window, pull down the File menu and select New Movie Window.



What you get looks a little intimidating at first, but all a movie window really is is a fancy turtle window. You can draw in the movie window the same way you draw in a turtle window. Give is a try by typing

```
SETPC 4
FLAG 20
```

Let's stop for a moment and think about how movies work in a movie theater. What you are really seeing on the screen is a series of still pictures, shown one after the other. If they are shown quickly enough, and if the difference between each picture is small and continuous, your brain tells you you're seeing motion. That's the way the movie window works, so what we need to do is create a few more frames.

To add the next frame, select the movie window, then pull down the Movie menu and select Add Frame After. The flag you just drew disappears! If you look down at the bottom of the window, though, you'll see a 2. That tells you you're on frame 2, which doesn't have a drawing, yet. Click on the « button, and you'll move back to frame 1, where the flag is still safe and sound. Click on the » button, and you'll be back at the new frame.

The movie we're going to create is a pretty simple one, with the flag twirling around in a circle. To draw the second frame, type

```
LEFT 45
FLAG 20
```

Add one more frame, but notice where the turtle is. It's still rotated 45° to the left. That's an important characteristic of the movie window we'll use to our advantage, now, as we add the next six frames of the movie.

```
REPEAT 6 [FLAG 20 LEFT 45 ADDFRAME]
```

Our first movie is almost complete, but there is one last step. Our repeat loop left us with a blank frame at the end of the movie. To get rid of it, you can either type

```
DELETEFRAME
```

or you can select the movie window, pull down the Movie menu, and select Delete Frame.

Playing the Movie

The two controls we haven't used are play and stop. To play your movie, select the movie window, then click on the play button, ▶. To stop the movie, click on the stop button, ⏸.

Exploring This movie only has 8 frames, so it's a little jumpy. You can create a movie with more frames, and get smoother motion, by using `360 / :frames` for the rotation angle, where `frames` is the number of frames you will use. The movie will draw a lot faster if you stick to integer angles and hide the turtle first.

For example, here's the commands to create a movie with 18 frames:

```
HIDETURTLE
SETPC 4
REPEAT 18 [LT 20 FLAG 20 ADDFRAME]
DELETEFRAME
```

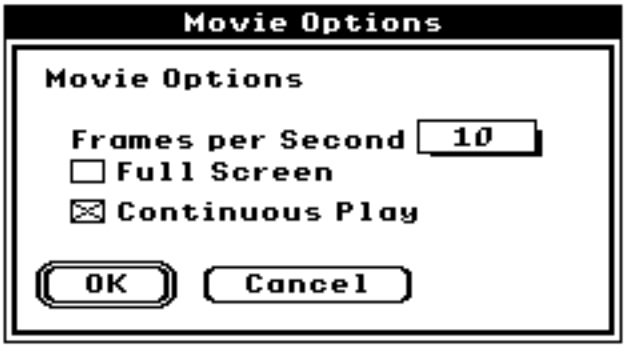
Of course, you'll want to do this in a new movie window. Be sure and close your old movie window first, too.

At some point, you'll run out of memory as you try to add more frames. Movies take lots and lots of memory. That's why it's a good idea to close all of your movie windows before you start a new one.

To get an idea just how many frames you can create with the memory you have, keep cranking the number of frames up in this example until you get an out of memory error.

Movie Options

Click on your movie window, then pull down the Movie window and select Movie Options... You'll see a dialog like this one.



Frames per Second controls the speed of the movie. You pick the number of frames you want the computer to show each second. The movie player can go as fast as 30 frames per second, which is the same speed your television uses. To get a full 30 frames per second, though, you'll have to keep the movie window small, like it starts, and you'll have to have an accelerator card. If the computer isn't fast enough to draw the frames as fast as you want, Logo will skip frames to keep the movie playing at the correct overall rate.

The big problem with a fast frame rate isn't the speed of the computer, though, it's the amount of memory you have. If your goal is a 3 second movie, and you're using the default rate of 10 frames per second, you'll need 30 frames – about 1 megabyte worth of pictures. If you pick 30 frames per second, to get the same 3 seconds, you'll need three times as much memory.

Below 10 frames per second, most pictures look so jumpy that they look more like a fast slide show than a movie. That's OK, though – these slow rates are handy when you're fine tuning a complicated movie.

When you're playing a finished creation, you might not want to see all of the Logo windows in the background. Try picking Full Screen, then click OK, and finally, click the play button in the movie window. What you get is a full screen movie.

Of course, the Stop button is gone, so there's no obvious way to stop the movie. Holding down the open-apple key and pressing period (⌘.) does the trick, though.

The last option is Continuos Play. When this option is selected, the movie starts playing again as soon as it is finished. This is a great way to get a lot more time from a simple movie. When you first ran the flag movie, for example, you probably didn't notice that it took less than one second to run! You watched a continuos movie until you got tired of it, then went back to the book.

Movies and Other Programs

You can load and save movies just like you can load and save the contents of any other Logo window. Logo stores movies on your disk using the popular Painworks animation format. That means you can load and play Logo movies with just about any movie player, and you can load and play a lot of the movies you'll find on online services in Logo.

Movie Samples

You already know everything you need to know to create movies with Logo. Before moving on, though, let's take a look at a few other examples that show how versatile movies can be.

▲ **Warning** Some of these movies use a lot of memory. You may need to change the number of frames they create to get the movies to work on your computer. ▲

The Cube Movie

The cubes movie shows two cubes rotating at different speeds. The cubes are in three dimensions, too. To create the movie, load the file CubeMovie from your samples folder and type

```
CubeMovie
```

This movie shows you one way to handle multiple objects in the same movie. In this example, the movie is completely created with one of the two cubes. The program then backs up to the start of the movie and moves through the frames again, drawing the second cube. That way, you only have to keep track of one object at a time.

This movie also shows how well movies and the 3D display work together. Movies or 3D separately are powerful tools for showing an object in space, but when they are combined, the effect is a lot more realistic. Once you add motion, even lines and points get easier to see in three dimensions.

Spinning Octahedrons

This movie shows two octahedrons. It uses the same technique as the cube movie to show two different objects in the same movie. This movie shows two new kinds of motion, though.

To create the movie, create a new movie window, load OctaMovie from your samples folder, and type

```
OctaMovie
```

All of the movies you've created before this one are made by rotating an object around some point, so the object sits at one place in space and turns. The larger of the two octahedrons in this movie does the same thing, but it's on a tether, so it's rotating in a circle around a center point it never touches. It's as if you spun the object from an invisible string.

The second octahedron doesn't spin at all. Instead, it moved through the picture, diving through the circle created by the first octahedron. It shows a good way to handle motion, starting from a point off of the screen, moving through the field of view, then vanishing off the other end of the screen. (If you make the movie display too large, you can see the entire path – the idea is to make the movie window big enough to see the circle formed by the first octahedron, but no bigger.)

Spinning Molecules

This time, you'll create a movie from an older example. Start by closing all of the windows except the text window you are typing in, and clear the workspace with

```
ERALL
```

Open a new movie window, then load the chemistry example from the last chapter with

```
LOAD "Chemistry
```

You already know how to draw a molecule. With a complicated one like $C_3O_3H_8$, though, it's tough to see the exact relationship of the different atoms. A movie makes it easy. Here's all you need to create a movie of the molecule:

```
RT 30
REPEAT 18 [HT Draw "C3O3H8 RLL 20 ADDFRAME]
DELETEFRAME
```

Exploring You may already see some of the possibilities for movies. One is to show chemical reactions. Instead of rotating a molecule in the middle of the screen, you can move two molecules from off of the screen to the center, flash a red screen, then move the new molecules that are created in the reaction off of the screen.

Chapter 5 – Talking Logo

What You Need

If you have our Talking Tools software package, Logo can talk. Anything you can type, Logo can read back to you using standard English pronunciation. If you don't have Talking Tools, Logo ignores all of the commands in this chapter. It won't do any harm to use them, but the commands won't do anything, either.

If you haven't installed talking tools, one way to install them is to run the Installer from Logo and pick the script "Install Talking Tools".

Making Logo Talk

Logo's SAY command will take any word and say it. It's really that simple.

```
SAY "I\ am\ completely\ operational,\ and\ all\ my\ circuits\
are\ functioning\ perfectly.
```

Teaching Logo to Pronounce Words

The voice sounds mechanical, but most of the words are pronounced clearly. There is one exception, though. When the computer reads "circuits", it sounds more like it is spelled – serkewits. A pair of commands sets this straight. `PHONETIC` takes a word as input and converts it to special phonetic symbols. You really don't need to worry about the phonetic symbols much, since you will almost always feed the output from `PHONETIC` straight to the `DICT` command, which is a phonetic dictionary. The `DICT` command takes two words as input. The first is the word you want to teach Logo to say, and the second word is the correct phonetic spelling for the word. The practical way to put these commands to use is to spell the word the way it sounds, using the `PHONETIC` command to convert this to phonetics, then pass the result straight to `DICT`.

Here's a line that will teach Logo to say circuits properly.

```
DICT "circuits PHONETIC "serkits
```

Try this, then ask Logo to read the original line again.

Exploring While you probably don't need to know about the phonetic symbols used by Logo, you might enjoy learning about them. Using phonetic symbols, you can control the way Logo talks

in very subtle ways. The phonetic symbols used by Logo are described in the Talking Tools manual. ☎

Male and Female Voices

By default, Logo uses a male computer voice. It's easy enough to switch genders:

```
VOICE "female
SAY "My\ circuits\ are\ doing\ fine,\ too!
VOICE "male
SAY "I\ noticed.
```

Speed, Volume and Pitch

Three other commands fine tune the voice. For all three commands, you can use a value of 1 to 9, and the default is 5. If you use a number outside of this range, Logo pins the value to 1 or 9, whichever is closest.

SPEED controls how fast Logo reads the words. The larger the number, the faster Logo talks.

```
SPEED 9
SAY "That\ was\ fast!
```

PITCH changes the pitch. A higher number raises the pitch, making the voice sound a little higher. Raising the pitch of the male voice, for example, makes it sound a little more like a female voice.

VOLUME controls how loud the voice is. With a higher volume, Logo shouts louder, and with a lower one, the voice is quieter.

Chapter 6 – Desktop Programs with 3D Logo

The Coloring Book Program

This chapter shows you how to create desktop programs with 3D Logo. You'll do that by exploring a desktop coloring book program, learning the steps as you go.

With any visual program, the first step is to figure out what you are trying to write, and desktop programs are certainly visual programs. We have a pretty specific program in mind, so the easy way to see what you'll be writing is to run the finished program. That's not something you generally get the chance to do in real life!

From 3D Logo, pull down the File menu and pick Run a Program... This command looks and works pretty much like the Open... command, but it has some special qualities. It only lets you load a specific kind of file, and once it does, it runs the Logo program right away. In the process, your old workspace is erased, so you'll get a prompt warning you ahead of time.

Tip These program files are special in another way, too. You can double-click on programs from the Finder, and the Finder will run 3D Logo, which will run the program and quit. That means you can write programs with Logo that you can run from the Finder.

When the program starts, you'll see a colored bar across the top of the screen. This colored bar is a set of paints. Pull down the File menu and pick Open, then choose Shapes. Click in the window, and the place you click gets filled in – painted – with the current color. Of course, you can switch colors, too. You have to click on the paints window first to select it, then click on the color you want. It will be outlined to show the color is selected. Click on the picture again to select the picture window, then click inside the window to paint something else.

This is a simple program, but it does quite a lot. You can open as many windows as you like. You can resize the windows, scroll from place to place, and save the painted pictures you create. You can even print the pictures, and they will print in color if your printer can handle color.

Logo Programs

Desktop programs don't have to be handled as programs, like the coloring book program you just ran, and you don't have to write a desktop program if you want to create a program you can launch from the Finder. It just happened that you're seeing these two things for the first time when they are used together.

There are only two requirements for a Logo program. The first is that it has to have a procedure called `Program`. This is the procedure Logo will run after the program is loaded into the workspace. This procedure does not have any parameters.

The other requirement for a program is that you have to save it to disk using the Save a Program... command. It works just like the Save As... command, but saves the file using the special Logo file type for Logo programs.

The Event Loop

Desktop programs spend most of their time waiting for you to do something. When you click on something with the mouse or type something on the keyboard, the program leaps into action, doing whatever the programmer decided was appropriate. As soon as it finishes, the program comes back and waits for you to do something else. Programmers call the things the program waits for – keypresses, mouse clicks and the like – events. The part of the program that waits for the events is called the event loop.

In the coloring book program, the event loop is in the `Program` procedure that gets called when the program starts. Here's that procedure.

```
TO Program
! [Main program]
DESKTOP
CreateMenus
CreatePaints
MAKE "Done "FALSE
EVENTS NOT :done []
ClosePaints
END
```

Comments

There are several things in this procedure that really don't have much to do with event loops. The first is the very first line after `TO`, which is a comment:

```
! [Main program]
```

Comments let you put little notes in a program to remind you what's going on at some future date. This one just says the procedure is the main program. The interesting thing about this comment, though, is that Logo doesn't have a statement for comments. Inside the program, you'll also find this procedure:

```
TO ! :p
END
```

The `!` procedure takes whatever you give it as input and ignores it. You can use a list, which holds text comments very nicely.

Starting Your Desktop

The next line calls Logo's DESKTOP procedure. DESKTOP sets up a Logo program. It hides all of the normal Logo windows, erases Logo's menu bar, and leaves you with a blank desktop and a blank menu bar. If you start in 640 mode, DESKTOP switches to the 320 mode screen.

DESKTOP a little different than the other Logo procedures you've used, since it can only be used inside of a procedure. As soon as the procedure finishes, Logo automatically cleans up the desktop, redraws the Logo menu bar, and redraws all of Logo's menus.

Exploring There are three variants on the DESKTOP command you might want to try. The first is GRAPHICS, which creates a screen with a single huge turtle window. It's one way to draw on the whole screen. Both DESKTOP and GRAPHICS start in 320 mode, but there are 640 mode versions of each procedure, too. The 640 mode versions are called DESKTOP640 and GRAPHICS640.

The Event Loop

There are some initialization and shutdown calls that we'll ignore for now. The event loop itself is just these two lines:

```
MAKE "Done "FALSE
EVENTS NOT :done []
```

The EVENTS command is really pretty simple. So far, it's just an infinite loop. EVENTS evaluates the condition NOT :done, and keeps right on evaluating that condition until it is false. Meanwhile, it checks for mouse clicks and key presses, handling them if it needs to. You can stuff a Logo statement in the list at the end, just like you do for the REPEAT statement, or leave it blank. If there are any commands in the list, they get executed anytime EVENTS doesn't have an event to handle.

Of course, looking at these two lines, you can see that :done will always be false, so NOT :done will always be true, and the program will keep chugging along until we stop it. That's true, so far, since we haven't created any menus or windows to do something when an event is found.

Tip If you get stuck like this in a program, and just want to get out so you can make changes, hold down the open apple key and press period (.). This always stops a Logo program.

Creating a Menu Bar

The first sort of event we'll handle is a menu event, when the user picks a menu command. Creating and using menu bars is really pretty easy, but the program looks a little messy. Rather than jumping right in, we'll build up the idea of menus from the bottom up, then look at the part of the program that creates them.

Menu Items

Basically, the whole idea behind a menu bar is to put some simple commands up on the screen. The user pulls down a menu and picks one of the commands. There are three things we need to tell Logo about each of the menu commands:

- The name of the menu command.
- Any optional things we want to use. The example we'll look at right away is a keyboard equivalent. Using ⌘Q for quit is one example of a keyboard equivalent.
- What we want to do when the user picks the menu command.

Collected together, these three things form a menu item. Since Logo is a list processing language, it's pretty natural to stuff these things into a list. Here's the Quit menu item from the coloring book program:

```
[Quit \*Qq [MAKE "done "TRUE]]
```

The first thing in the list is the name of the menu command, which Logo will print in the menu exactly as you type it.

The next part is the options. `*Qq` tells Logo you want to use a keyboard equivalent. The * character signals that you're defining a keyboard equivalent, and the two characters that come right after it are the two keys that Logo will accept. If you hold down the open-apple key and press either keyboard equivalent, Logo acts like you selected the command from the menu bar.

Of course, * happens to be a Logo separator, so you need to put a \ character in front of it.

▲ **Warning** Logo is pretty sensitive about key equivalents and other options. You must use exactly two characters for a key equivalent. The first of the keys is the one that will be displayed in the menu bar. By convention, the first letter is an uppercase character, and the second one is it's lowercase equivalent. ▲

The last thing in the menu item is a list. You can use the empty list, and you'll even do that a little later. In this case, you finally see how the program stops. When the user picks the Quit menu command, Logo executes the line

```
MAKE "done "TRUE
```

Since `:done` is finally `TRUE`, the `EVENTS` command will stop checking for events, and the program will stop.

☞ **Exploring** There are several other menu options you might want to explore. Look at the MENUBAR command in Chapter 8 for details. ☜

Menus

All of the menu commands are collected in menus. Each of the menus has a name at the top of the menu bar, and shows some menu items when you pull down the menu. In Logo, the menu bar is another list. The first thing in the menu bar is the name of the menu. All of the menu items are lined up after the menu name, in the order they appear when you pull down the menu.

It's easy enough to type in a menu as a list, but it gets to be very long, and that makes it hard to read. In the coloring book program, you'll see a different idea. Instead of typing in a huge list, the menu is built up item by item using the `LPUT` command, which adds a new object to the end of an existing list. This way, each menu item is on a separate line, so you can move the items around or add or delete menu items with very simple editing commands. With a little practice, you can even "see" what the menu will look like by scanning the list of commands.

Here's the File menu from the coloring book program:

```
! [Set up the File menu.]
MAKE "Menu [\ File\ ]
MAKE "Menu LPUT [Open... \*Oo [DoOpen]] :Menu
MAKE "Menu LPUT [Close N255\*Ww [DoClose]] :Menu
MAKE "Menu LPUT [Save \*Ss [DoSave]] :Menu
MAKE "Menu LPUT [Save\ As... [DoSaveAs]] :Menu
MAKE "Menu LPUT LIST :Dash [] :Menu
MAKE "Menu LPUT [Page\ Setup... [DoPageSetup]] :Menu
MAKE "Menu LPUT [Print... [DoPrint]] :Menu
MAKE "Menu LPUT LIST :Dash [] :Menu
MAKE "Menu LPUT [Quit \*Qq [MAKE "Done "TRUE]] :Menu
```

Once these commands finish, `Menu` contains a list which starts with a menu name, and has a list for each menu item right after the menu name.

Note If you want to see the actual list, use the `PO` command to list the procedure `CreateMenus` in your text window, then select these lines and run them by pressing the return key. That creates the list. You can look at the list using the `SHOW` command.

Looking through these commands, there are several interesting things to notice.

First, most of the commands just call some other Logo procedure. That's a nice, simple way to organize the menus.

The name of the menu looks a little peculiar. For the menu items, we just used the same characters that we wanted to show when the menu was pulled down. For the menu itself, though, the name is `\ File\ `. The backslash characters are being used to tack a space on either side of the file name. That's important, since you want some separation between the names of the menus, and Logo doesn't do that automatically.

The last odd thing about this menu is the two menu items formed with `:Dash`. The variable itself is a word, set up earlier in the procedure with this line:

```
MAKE "Dash "-
```

There are two peculiar things here. First off, a menu with a name that is just a single minus sign is treated in a very special way. The Apple IIGS menu manager creates a dividing line from this menu item. That's the line you see across the menu bar when you pull down the menu. It's disabled, too, so you can't select this particular item. The other odd thing about this menu item is that we can't create it the same way as the others, since Logo treats the - character in a special way, too. Instead of using `LPUT` and a list we type in, our program creates a list with the `LIST` command. The command

```
LIST :Dash []
```

returns the list

```
[\- []]
```

which becomes the menu item for the separator. It looks a little weird, but it's just a menu item with a name of - that doesn't do anything when you select the command. That's OK, since you can't pick this menu command anyway.

Exploring

You're probably pretty used to looking in the File menu for Quit, using ⌘O as the key equivalent for the Open command, seeing ... after menu commands that bring up a dialog, and a lot of other little standards that seem to be in just about every desktop program. The reason these things seem so standard is, well, they are.

Apple Computer has developed and published a set of guidelines for writing desktop programs. These guidelines are in a book called Apple Human Interface Guidelines. If you're going to create programs other people will use, this book is one you really must have. Even if you don't want to write programs for other people, though, I'd suggest getting this book and reading it. It's a gold mine of tips about programs

that run on Apple computers. Lots of times, programmers follow these guidelines and just expect you know that features exist, since they are a part of the standard. For example, this manual never describes shift-clicking to extend a selection – it just assumes you know about it. The Apple Human Interface Guidelines will tell you about all sorts of shortcuts like this that will work in a lot of programs you use every day.

The Menu Bar

A Logo menu item is a list, and a Logo menu is a list with the menu name and menu item lists, so you're probably expecting the next step: A Logo menu bar is a list of menus.

If you haven't already opened the Logo workspace to look at the coloring book program, do that now. Inside the `CreateMenus` procedure, you'll find some lines that are using `LPUT` to build the menu bar, just like we used `LPUT` earlier to build a menu.

```
MAKE "Bar []
...
MAKE "Bar LPUT :Menu :Bar
```

Logo's `MENUBAR` command takes the finished menu bar list and draws the menu bar.

```
MENUBAR :Bar
```

Desk Accessories

You know enough now to create a Logo program that handles menu commands, but there is one other thing you'll want to do in almost any desktop program. Almost all desktop programs start with an Apple menu, and the coloring book program is no exception. Creating the Apple menu isn't hard, but there is the little problem that there isn't a colored apple key on the keyboard. Logo solves this problem by treating a menu with the name @ as a special case, drawing an Apple menu.

The menu items in an Apple menu are special, too. The first one usually brings up an about box. We'll talk about that in the next section. Under the About... command, though, you generally see a list of desk accessories. It isn't hard to support desk accessories in your Logo programs. In fact, there are only two steps you need to take.

First, desk accessories use menu commands, too. They expect six menu items to be set up in a special way. All of these menu items have an extra option that sets something called a menu item ID. That's not something you generally have to worry about in Logo desktop programs, but for these six menu items, you have to put it in. The menu ID is the character N followed by a number, and the number you use is very important. Here are the six menu items you should have in every desktop program that supports desk accessories, whether your program uses these items or not:

```
[Undo N250\*Zz []]
[Cut N251\*Xx []]
[Copy N252\*Cc []]
[Paste N253\*Vv []]
[Clear N254 []]
[Close N255\*Ww []]
```

Desk accessories really don't care where you put these menu items, as long as they exist. Desk accessories don't care whether you use these keyboard equivalents, either. You're used to seeing these menu items in specific places in the File and Edit menus, though, and you're probably used to these key equivalents. The easiest way to create these menu items is to copy `CreateMenus` from the coloring book program into your own program. That way, these special menu items will already be in the right place in your program.

The other step you have to take to support desk accessories in your Logo programs is to call `ADDNDA` once, right after you create the menu bar with the `MENUBAR` command. `ADDNDA` attaches all of the desk accessories it finds in your system folder to the first menu in your menu bar. That's usually the Apple menu.

You'll find the call to `ADDNDA` at the end of the `CreateMenus` procedure in the coloring book program.

▲ **Warning** Never call `ADDNDA` until after you have set up the menu bar with `MENUBAR`. ▲

The About Box

You can create three different kinds of windows with Logo, and the coloring book program uses them all. All of the windows are created with Logo's `NEWWINDOW` command. Here's `DoAbout`, the procedure called when you pick About... from the apple menu.

```
TO DoAbout
! [About box]
LOCAL "AboutDone
NEWWINDOW [About 1 0 [40 136 280 76] [MAKE "AboutDone "TRUE] []]
DrawAbout
MAKE "AboutDone "FALSE
EVENTS NOT :AboutDone []
Ignore CLOSEW "About
END
```

The part that actually creates the window is the call to `NEWWINDOW`. `NEWWINDOW` expects a list with six objects:

field	example	description
name	`About`	The name of the window. If the window has a title bar, like most of them do, this is the name Logo will put in the middle of the title bar. It's also the name you use with the commands that work on a window. We'll look at some of these commands in a moment.
kind	`1`	This is the kind of the window. The about box is an alert, which is a simple box with a heavy line inside. If you use 2, you'll get a turtle window, like the drawing window. Using 3 for the kind gives you a turtle window with no scroll bars, like the paints window.
flags	`0`	Use a 1 for flags if the window holds something that gets saved to disk. When you use a 1, Logo will give the user a chance to save changes anytime a window is being closed without saving the changes to disk. The about box is just an information window, so it uses a 0, telling Logo it doesn't have to check before closing the window.
rect	`[40 136 280 76]`	This is a rectangle, which is a list that tells Logo where to draw a box. In the order they show up in the rectangle, the numbers are the left, top, right and bottom coordinates for the box. The box itself is the drawing area you'll have available, and it's located on the screen as if the screen is a turtle window with [0 0] at the bottom left of the screen.
click action	`[MAKE "AboutDone "TRUE]`	This list gets executed if the user clicks the mouse in your window while the window is the frontmost window. If the user uses the mouse to select the window, drag it, resize it or scroll it, Logo handles the event without bothering you.
key action	`[]`	This list gets executed when the user presses a key.

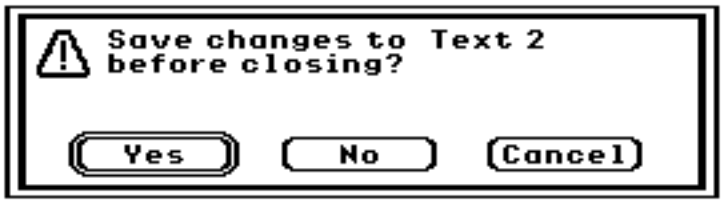
Each window you create is a turtle window. For a windows like this one that don't have a scroll bars, [0 0] is in the middle of the screen. All of the turtle commands you've learned will work just fine.

The rest of the `DoAbout` procedure handles the window created by `NEWWINDOW`. `DrawAbout` is a procedure in the program that draws the stuff in the about box. It uses a few font commands to create a cool window, but there's nothing special about it. We won't go over it here, but you can explore `DrawAbout` on your own.

The about box is a special sort of window called a dialog. To handle the dialog, we create a mini event loop that handles everything – and ignores everything – until we click in the window. Once that's done, `DoAbout` closes the window with the `CLOSEW` command.

```
Ignore CLOSEW "About
```

CLOSEW closes a window, returning true if the window actually got closed, and false if it didn't. The only way a window won't get closed is if it can be saved to disk, and it's been changed since the last time it was actually saved. In that case, the user sees this dialog:



If the user picks Cancel, the window doesn't get closed, and CLOSEW returns false. The about box uses 0 for the flags word, though, so the window will always get closed. Ignore works like the comment procedure; it just takes the value CLOSEW returns and does nothing with it.

The Paints Window

Way back at the start of this chapter, when we looked at the Program procedure, there were two procedure calls we ignored. These procedure calls were for CreatePaints and ClosePaints, to create and close the paints window. Here are the procedures, along with PaintsClick, which handles clicks in the paints window.

```
   TO CreatePaints
   ! [Create the paints window]
   NEWWINDOW [Paints 3 0 [0 187 320 167] [PaintsClick] []]
   LOCAL "r
   LOCAL "p
   MAKE "color 0
   MAKE "p -160
   REPEAT 16 [MAKE "r [:p 10 :p + 20 -10] SETPC :color PAINTRECT
:r MAKE "p :p + 20 MAKE "color :color + 1]
   MAKE "color 0
   FrameColor
   END

   TO PaintsClick
   FrameColor
   MAKE "color INT (FIRST MOUSE + 160) / 20
   FrameColor
   END
```

```
TO ClosePaints
Ignore CLOSEW "Paints
END
```

These procedures are a little longer than most of the ones in this book, but breaking them down, there isn't much here that's new. `CreatePaints` creates the paint window pretty much the same way `DoAbout` created the about box. The only real difference is that this window has a kind of 3, which is a turtle window with no scroll bars. The window actually does have a title bar, which is the place above most windows with the name of the window and a close box. The reason you don't see it is due to some careful positioning: The title bar for this window is actually hidden behind the menu bar.

There's a fair amount of math inside these procedures. All the program is really doing is dividing the window up into 16 blocks, one per turtle pen color, then filling each block with a different color. When you click in the window, `PaintsClick` uses the same idea to figure out which box you picked, setting the variable `:color` to the color you want to use. Later on, we'll read this color to decide what color to use to fill in an area in the drawing window. `FrameColor` is the procedure that draws the outline box that shows which color you've picked.

Drawing Windows

The last kind of window is a full-blown turtle window with scroll bars. The coloring book program uses this kind of window for the drawing window you actually color in.

Turtle windows look and work just like the turtle windows you've used all along. The drawing area in each window is as large as one screen. You use scroll bars to move around in the window, since you can't see the entire picture at once.

The coloring book program opens a turtle window when you pull down the File menu and pick Open... The Open... command calls `DoOpen`:

```
TO DoOpen
! [Open a new window]
NEWWINDOW [Untitled 2 1 [5 149 295 18] [DoClick] []]
IF NOT LOADW [] [Ignore CLOSEW []]
END
```

The `NEWWINDOW` command works just like the ones for the alert window and paints window. The only new thing in this procedure is the call to `LOADW`, which loads a file from disk. It uses an Open dialog, just like the one Logo uses when you use Logo's Open... command to open a document. When the user picks a file, the contents get loaded into the turtle window. If something goes wrong – if the user picks cancel, or if there is some sort of disk error – `LOADW` returns false, and our program closes the window that we just opened.

The parameters to `LOADW` and `CLOSEW` may seem a bit strange. When you give one of the window commands an empty list instead of a window name, it uses the front window. That's a

convenient shortcut for us, since we know the window we want to work on is the front window – we just opened it, after all.

When you click in the coloring window, DoClick handles the click.. It moves the turtle to the spot where you clicked, sets the pen color to the color you picked in the paints window, and uses Logo's FILL command to fill in the area with a color.

```
TO DoClick
PU
SETPOS MOUSE
PD
SETPC :color
FILL
END
```

Commands That Effect Windows

The rest of the program is very simple. The other commands that the program handles are Close, Save, Save As..., Page Setup... and Print... In each case, there is a single Logo procedure you can use to do everything that needs to be done. All you have to supply is a window name, and for all of these commands, you just use the front window. The only trick is to use WINDOWP to make sure there is an open window before you use the command.

```
TO DoClose
! [Close the front window]
IF WINDOWP [Ignore CLOSEW []]
END

TO DoSave
! [Save the front window]
IF WINDOWP [SAVEW []]
END

TO DoSaveAs
! [Save the front window to a new file]
IF WINDOWP [SAVEASW []]
END

TO DoPageSetup
! [Get printing options]
IF WINDOWP [PAGESETUPW []]
END
```

```
TO DoPrint
! [Print the front window]
IF WINDOWP [PRINTW []]
END
```

☕ **Exploring** The coloring book program is a pretty typical desktop program. There are a few commands and options this program doesn't use, though. To find out more about desktop programming, browse through "Desktop Programs" in Chapter 8. ☕

Chapter 7 – Desktop Interface Reference

This chapter describes the desktop interface used by 3D Logo. Each of the menu commands is described. See Chapter 8 for a detailed description of the Logo language.

After an initial discussion of the Finder interface, this chapter is organized by menu, with each command from each menu listed in order under the menu heading.

There are four kinds of windows used by Logo. The use of the windows is described under the various New commands. The new commands, in turn, are described under the File menu.

In several places, this documentation refers you to Apple's System Disk manuals. These are the manuals that came with your computer. The most recent version comes with Apple's System Disk 6.0, available from a variety of sources.

Finder Interface

Teaching the Finder About Logo

Apple's Finder can automatically use Logo to open files, print documents or run Logo programs, as described in the next few sections. Before it can do that, though, you have to tell the Finder Logo exists, and just which files it can use. Teaching the Finder about Logo is pretty simple, though – all you have to do is run Logo3D.Sys16 one time from the Finder. When you do that, the Finder saves some information about Logo in it's desktop file, and from then on (unless you delete the desktop file, of course), the Finder can do all of the things described in the next three sections.

Opening Files

When you open a 3D Logo document from Apple's Finder, either by selecting one or more documents and opening them or by double-clicking on a Logo document, the Finder will start 3D Logo, which will open a window for each of the documents you picked.

Running Programs

Logo has one special case. When you open a Logo program file, Logo runs a program. Logo program files are files with a file type of $B0 and an auxiliary file type of $0C12. They are generally created with Logo's Save a Program... command, described later in this chapter. Most Logo programs will be desktop programs, like the ones described in Chapter 6.

When you open a Logo program file, Logo ignored all other files that you open. The Logo program is loaded into the workspace, and Logo calls the procedure PROGRAM. As soon as the program finishes running, 3D Logo quits.

The effect, then, is that you can click on a Logo program file from the Finder, and the program runs, just as if it were a normal system program.

Printing Files

You can print Logo files without running Logo. Select the files you want to print and pick the Print command from the Finder's menu. The Finder runs Logo, which prints the files you selected and returns to the Finder.

Apple Menu

About 3D Logo...

The About alert shows the copyright notice and version number for 3D Logo. The version number is something you should check, especially if you are about to call for assistance with 3D Logo.

Desk Accessories

Any remaining items in this menu are desk accessories. You can find a general description of desk accessories in Apple's System Disk manuals. For specific information about a particular desk accessory, see the documentation that comes with the desk accessory.

File Menu

New

The New command opens a new text window, like the one that opens by default on the right side of the screen. A text window is a standard edit window, as described in Apple's System Disk manual, with one twist: The RETURN key has special meaning. When you press the RETURN key, Logo executes the line the insertion point is on, writing any output at the end of the window.

If you select some text, then press RETURN, Logo executes all of the text, even if you have selected several lines.

With that exception, text windows are just TextEdit windows. You can scroll them, change the size of the window, select text, or edit text using the standard editing commands described in Apple's System Disk manual.

When you save a text window, Logo saves the file as an Apple IIGS language file. It uses a file type of $B0 (SRC) and an auxiliary file type of $0112 (Byte Works Logo).

Logo opens a text window any time you use the Open command to open any form of ASCII file.

New Edit Window

Edit windows work pretty much like text windows created with the New command. There are two differences.

First, the RETURN key is not treated in a special way. The RETURN key simply splits the line, moving to the start of the new line.

You'll see the second difference when you close the edit window or select some other window. At that time, Logo runs all of the commands in the window.

Generally, you'll open an edit window with Logo's `EDIT` or `EDITS` command. Once the window is open, you can make changes to the procedures or variables you are editing, then switch back to the text window to test the changes.

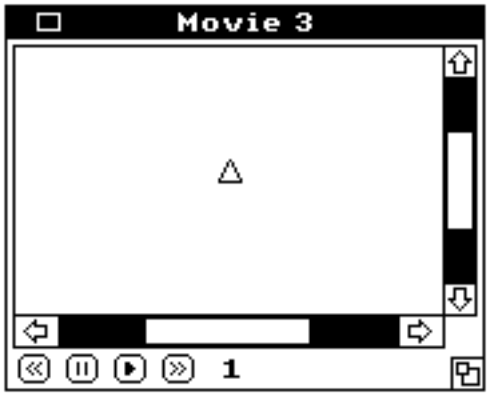
New Turtle Window

This command opens a new, blank turtle window, like the one that shows up on the left side of the screen when you run Logo. The window shows a portion of the complete drawing area, which is as large as the entire Apple IIGS screen. You can use the scroll bars, size box, and zoom box to look at the various parts of the picture.

When you save a turtle window, Logo uses a graphics format supported by most paint programs. Logo can open any Apple Preferred format picture, which you can use as a background for your Logo drawings.

New Movie Window

This command opens a new movie window, like this one:



The drawing area (the rectangle above and to the left of the scroll bars) looks and works exactly like a turtle drawing window. All of the Logo drawing commands work the same way.

The total size of the drawing area is one computer screen. A computer screen is 200 pixels tall, and either 320 pixels wide or 640 pixels wide, depending on the drawing mode. The scroll bars can be used to display different parts of the screen, and the grow box can be used to enlarge the drawing area.

Note A full screen display is also available for the movie window, but only when you are playing a movie. See "Movie Options...", later in this chapter, for details.

The four buttons along the bottom of the window are used to move from one frame to another, play a movie, and stop a movie.

Last Frame	Move to the frame before the one that is displayed. If the movie window is showing the first frame, this button is ignored.
Stop	If a movie is playing, this button stops the movie. If no movie is playing, this button is ignored.
Play	Start playing a movie. Once a movie starts, the only command the program will accept is to stop the movie. You can stop the movie by pressing the stop button or by pressing open-apple period (.).
Next Frame	Move to the frame after the one being displayed. If the movie window is showing the last frame of the movie, this button is ignored.

The number to the right of the buttons is the frame number. The first frame is frame number 1. The frames are numbered sequentially.

There are several Logo commands which can add or delete frames, or mimic the operation of the buttons in the movie window. See "Movies" in Chapter 8 for details.

The Movies menu has four commands to add and delete frames and set various options that control how a movie is displayed. See "Movies", later in this chapter, for details.

Open

The Open command brings up a standard Apple open dialog. The open dialog is described in Apple's System Disk manual.

Logo can open three kinds of files.

Pictures	Apple preferred format pictures (file type $C0, auxiliary file type $0002) are opened as turtle windows.
Animations	Paintworks animation files (file type $C2, auxiliary file type $0000) are opened as movie windows.
Text	Text Files (file type $04) and program source files (file type $B0) are opened as text windows.

Logo won't let you make a mistake and open a file it can't handle. The only time you need to worry about the actual file type is when you are creating a file from another program that you want to load into Logo.

Close

This command closes the front window. It works with all Logo windows and most windows opened by desk accessories.

Save

Saves the contents of the front window. If the front window has never been saved to disk, this command works exactly like the Save As... command.

Save As...

This command brings up a standard Apple save dialog, which will let you pick a location and file name to use to save a file. Apple's save dialog is described in Apple's System Disk manual.

There are four kinds of windows in Logo, and each is saved in a slightly different way.

Text Window	Text windows are saved as Logo program source files. The file type is $B0, and the auxiliary file type is $0112.
Edit Window	Edit windows are saved as Logo program source files, too. The file type is $B0, and the auxiliary file type is $0112.
Turtle Window	Turtle windows are saved as Apple Preferred picture images. The file type is $C0, and the auxiliary file type is $0002.
Movie Window	Movie windows are saved as Paintworks animation files. The file type is $C2, and the auxiliary file type is $0000.

Run a Program...

This command is used to run a Logo program. It brings up a standard Apple open dialog, which will let you select any Logo program file (file type $B0, auxiliary file type $0C12). Logo program files are generally created with Logo's Save a Program... command, described in the next section. Apple's open dialog is described in Apple's System Disk manual.

Before loading a Logo program, Logo clears the workspace. You are warned before the workspace is cleared. A dialog will appear that gives you the option of continuing or canceling the command without clearing the workspace.

Logo programs must contain a procedure called PROGRAM. Once the program is loaded, Logo calls this procedure to run your program. When the program finishes, it is left in the workspace where you can edit, change, or re-run the program.

Save a Program...

This command brings up a standard Apple save dialog, like the one used for the Save As... command. Apple's save dialog is described in Apple's System Disk manuals.

Once you pick a file name, this command saves the workspace as a program file. Logo program files have a file type of $B0 and an auxiliary file type of $0C12. In most cases, you will use this command to save graphics or desktop programs that will be run from the Finder, or occasionally using the Run a Program... command.

Page Setup...

This command opens a printer page setup dialog. Logo supports any printer driver that conforms to Apple's standards. Apple's printer drivers are described in Apple's System Disk manual; for other printer drivers, see the documentation that came with the driver.

Print...

This command opens a print dialog. Logo supports any printer driver that conforms to Apple's standards. Apple's printer drivers are described in Apple's System Disk manual; for other printer drivers, see the documentation that came with the driver.

Quit

This command leaves 3D Logo. Before leaving, all windows are closed. If any window has changed since it was last saved to disk (or since it was created, if it has never been saved to disk), you will get a chance to save the file, not save the file, or stop shutting down the program.

Edit

Undo

Undo exists to support desk accessories. This command is not used by Logo.

Cut

Cut is used by desk accessories and by the text and edit windows in Logo.

In Logo's text and edit windows, Cut does nothing if there is no text selected. If text is selected, Cut removes the text from the window, leaving the insertion point at the spot the text was removed from. The text is placed in the scrap. From there, you can use the Paste command to copy the text back into a Logo window, or you can use a scrapbook desk accessory to move the text to another program.

Copy

Copy is used by desk accessories and by the text and edit windows in Logo.

In Logo's text and edit windows, Copy does nothing if there is no text selected. If text is selected, the text is placed in the scrap. From there, you can use the Paste command to copy the text back into a Logo window, or you can use a scrapbook desk accessory to move the text to another program.

Paste

Paste is used by desk accessories and by the text and edit windows in Logo.

In Logo's text and edit windows, Paste starts by deleting any selected text, as if the Clear command were used. Next, Paste copies any text in the scrap into the window. The text is copied, not removed, from the scrap, so you can paste the same text several times in a window, or you can paste the text into several different windows.

Logo's Copy and Cut commands change the scrap. Desk accessories can also change the scrap.

Clear

Clear is used by desk accessories and by the text and edit windows in Logo.

In Logo's text and edit windows, Clear does nothing if there is no text selected. If text is selected, Clear removes the text from the window, leaving the insertion point at the spot the text was removed from.

Select All

This command is used with Logo's text and edit windows. It selects all of the text in the window.

Movie

The movie menu is only active when a Logo movie window is the front window.

Add Frame After

A new, blank frame is added after the frame that is currently displayed, then the frame is advanced so you are viewing the new frame.

Logo's `ADDFRAME` command does the same thing as this menu command.

Insert Frame Before

A new, blank frame is added before the frame that is currently displayed. The frame counter does not change, so you are left viewing the new frame.

Logo's `INSERTFRAME` command does the same thing as this menu command.

Delete Frame

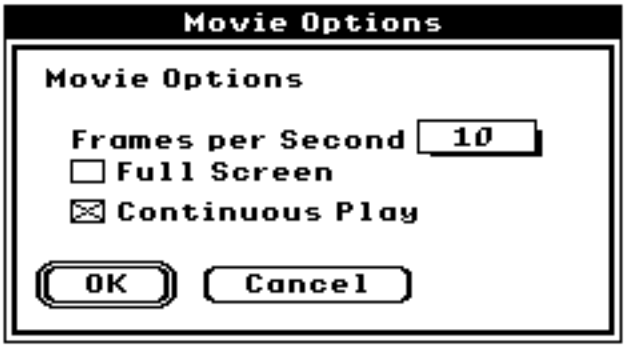
The visible frame is deleted from the movie.

This command is not available if the movie only has one frame.

Logo's `DELETEFRAME` command does the same thing as this menu command.

Movie Options...

This command brings up a dialog. You can use the dialog to select options that effect the way the current movie is played. The changes only effect the frontmost movie window.



Frames per Second 10

This pop-up menu lets you pick from a variety of preset play rates. You can play a movie as fast as 30 frames per second or as slow as one frame per second.

30 Frames per second is the same rate as a television set and most computer monitors. In some cases, the computer won't be able to keep up with this display rate, especially if your computer does not have an accelerator card, or if you are using full screen display and the picture changes radically between frames. Long movies with high speeds also take lots of memory.

The slowest frame rate of 1 frame per second is useful for checking your animations to see where the problems are.

For most cases, you should pick a frame rate from 7.5 to 15 frames per second.

☐ Full Screen

If this option is selected, movies will use the entire screen, including the area normally used by the menu bar. To stop a full screen movie, hold down the open-apple key and press the period key (⌘.).

☒ Continuous Play

If this option is selected, the movie will play continuously. Once the movie finishes, it loops back to the first frame and keeps playing. Carefully designed movies that loop back on themselves, like the rotation movies created in Chapter 4, can play continuously and look very good, yet use very little memory.

Windows

Use 640 Mode and Use 320 Mode

You can use this menu option to change from a 320 mode screen to a 640 mode screen and back again.

The 320 mode screen on the Apple IIGS is 320 pixels wide and 200 pixels high. Each pixel can be any one of 16 different colors. When you are using the 3D graphics display, 320 mode gives you black and three shades of gray. We recommend this mode when you are using Logo's graphics commands.

The 640 mode screen on the Apple IIGS is 640 pixels wide and 200 pixels high. Each pixel can be any one of 4 different colors, although alternating colors from one pixel to the next blends the four basic colors to give a total of 16 colors. When you are using the 3D graphics display, 640 mode limits you to black and white. We recommend this mode when you are primarily using text windows, and don't need colorful graphics.

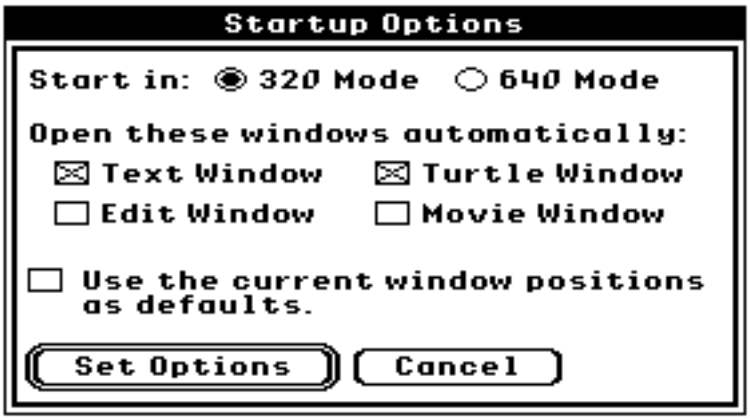
▲ **Warning** Pictures created in one graphics mode generally look pretty crummy in the other graphics mode. In most cases, you should pick a graphics mode and stick with it. ▲

△ **Important** By default, Logo sets the scrunch factor to 2 in 640 mode. The effect is that 640 mode acts like 320 mode, but with different colors, unless you change the scrunch factor. See `SETSCRUNCH` in Chapter 8 for details. △

Startup Options...

This command brings up the dialog you see below. You can use this dialog to set up your personal preferences for Logo. Once you set the preferences, Logo will use them whenever you start the program.

The preferences are stored in the file Logo3D.Options. If you booted from a file server, this file is saved in your user directory on the file server itself. If you did not boot from a file server, this file is saved in the same directory as the Logo program, Logo3D.Sys16. You can reset the factory defaults by deleting this file.



Start in: 320 Mode 640 Mode

If you select 640 mode, Logo starts in 640 graphics mode instead of 320 graphics mode.

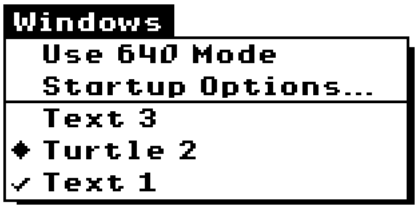
Open these windows automatically:

You can automatically open any of the four kinds of windows supported by 3D Logo. For each window type you pick, Logo will open one new window of that type each time you run the program.

Use the current window positions as defaults.

Logo has a standard size and position for each of the four kinds of windows it handles. You can pick new defaults for these sizes and positions. To do so. open one of each of the four kinds of windows, then position them and pick the sizes you want Logo to use. Select the windows you want Logo to open by default, then set the options. Logo will remember your preferences, and use them for any new window.

Windows by Name



All of your open Logo windows are listed right below the Startup Options... command. You can select a window, even if you can't see it, by picking the window from the list.

The window list also shows two other pieces of information. The active window – the one that will be saved, and if it's a text or edit window, the one that will get the results of your typing – has a check mark beside it. The current drawing window – the one Logo will draw in when you use any drawing commands – has a diamond.

Note The current drawing window is always the last drawing window that was a front window. So, to change the drawing window when you have more than one open turtle window or movie window, select the window you want to draw to. Selecting a text or edit window will not change the current drawing window.

Chapter 8 – 3D Logo Language Reference

Command Descriptions

This chapter is a technical reference to the Logo language. Each command is covered. The commands start with a description of the command that includes the name of the command, along with any parameters. Anything you have to type exactly as shown will be shown in uppercase letters. In the spots where you can substitute a value, you'll find a descriptive parameter in lowercase letters. If there are any punctuation marks, like [or] characters, they must be typed exactly as you see them.

Here's a typical example:

```
BUTLAST object
BL object
```

In this case, there are two names for the command, something that is very common in Logo. BUTLAST is a longer, descriptive name that tells you pretty much what the command does: it returns everything but the last element of a list, or all of the characters except the last one from a word. BUTLAST is used a lot, though, so to save typing, you can also use BL.

You'll find a detailed description of the command right after this header. The command description will tell you exactly what the command does, and what limitations it has.

If the command description doesn't show an example, the last thing in the entry for the command is a code snippet. The idea is to show you at least one example of the command in an actual line of Logo. This is usually just a single line of Logo, although a few useful Logo procedures are sprinkled here and there in the code snippets.

Finding Commands

There are three ways to find a command.

This chapter is organized by category, with commands for a category listed alphabetically within the category. That's a handy way to list the commands when you're looking for a command to do something, but you're not quite sure what the command will look like. For example, if you're looking for a drawing command, you can look in the turtle graphics section.

Scanning through the whole chapter is a little tedious if you have some idea what command you are looking for, but need to be reminded of the name. You might also remember the name, but want to check on exactly what the parameters are. Appendix A lists just the command headers, while the table of contents has an outline of this chapter by command and category.

Finally, if you're looking for the complete description of a command, and want to find it quickly, look in the index. All of the commands are listed there, and the starting page number for the technical description is shown in bold. Some commands are used in the tutorial section, too; you might find some useful tips there that aren't obvious from a technical description.

Procedures

COPYDEF

```
COPYDEF old-name new-name
```

COPYDEF creates a copy of a procedure. The new procedure has the same parameters and statements as the old procedure, it just has a new name.

Snippet

```
COPYDEF "Square "Polygon
```

DEFINE

```
DEFINE name list
```

DEFINE creates a new procedure. Once a procedure is created, it really doesn't matter whether it was created with DEFINE or with TO. The difference is that DEFINE works better when you want to create a procedure from inside another procedure, while TO works better if you are typing in a procedure in a text or edit window.

name is the name of the procedure.

list is a list of lists. The first element of the list is a list of parameters. If there are no parameters, you can use the empty list. The rest of the lists are the lines that make up the procedure.

For example, to create the procedure

```
TO Flag :size
FD :size
REPEAT 4 [FD :size RT 90]
PU
BK :size
PD
END
```

using DEFINE , you would use

```
DEFINE "Flag [[size] [FD :size] [REPEAT 4 [FD :size RT 90]] [PU] [BK :size] [PD]]
```

DEFINEDP

`DEFINEDP name`

`DEFINEDP` returns `TRUE` if name is the name of a procedure created with `TO` or `DEFINE`, and `FALSE` if it is not.

Snippet

```
IF NOT DEFINEDP "Flag [DefineFlag]
```

PRIMITIVEP

`PRIMITIVEP name`

`PRIMITIVEP` returns `TRUE` if `name` is the built-in procedures, and `FALSE` if it is not.

Note **Primitive** is the technical name for one of the procedures that are built into the Logo language. `PRIMITIVEP` gets it's name from this technical term.

Snippet

```
IF PRIMITIVEP :command [PR [Can't TO that!]]
```

TEXT

`TEXT name`

`TEXT` returns the procedure `name` as a list. The list uses the same format as the input list for `DEFINE`.

Snippet

```
PR TEXT "Square
```

TO

```
TO name parm1 parm2 ...
```

TO is a special command that lets you type in new procedures. The first line is a description of the procedure. It gives the name of the procedure, and lists the parameters. You don't have to include any parameters, but there is no limit to the number of parameters, either.

Here's an example of TO. This line defined a procedure called Greet that has a single parameter.

```
to Greet :who
```

The lines that actually get executed come right after the line with TO. END marks the end of the procedure. Here's what you would type to finish up the Greet procedure:

```
SHOW "Hi
SHOW :who
END
```

Once the procedure is in, you can use it over and over with different parameters, like this:

```
Greet "Fred
Greet "Sam
```

Parameters are local variables, but other than that they work just like any other variable. When you call the procedure, the value you type after the name of the procedure is assigned to the parameter. It works as if the procedure started out with the lines

```
LOCAL "who
MAKE "who "Fred
```

Other than available memory, there is no limit to the number of procedures you can define. There are also no restrictions on the way you can use your procedures. They work just like the built-in procedures, like SHOW. Of course, you can call your procedures from inside another procedure or from any button script, and you can even have a procedure call itself.

There is one restriction you have to keep in mind when you pick a name for your procedure: It can't have the same name as one of the built-in procedures. Other than that, anything Logo will accept as a word works as a procedure name.

Variables

LOCAL

```
LOCAL name
```

LOCAL creates a local variable. Local variables exist only inside of a procedure, and vanish as soon as the procedure finishes executing, but with that exception, they work just like any other variable.

Using local variables makes it easier to create procedures you can reuse, moving them from one stack to another, since they don't interact with the rest of your buttons in unexpected ways. Ideally, your procedures should only use local variables and parameters. If you follow that rule, the only problem you have to worry about when you move one of your procedures into a new stack is whether you already have a procedure by the same name.

Once you create a local variable, commands the deal with variables will find your local variable first. For example, let's say you create a global variable called Fred, and identify Fred as a dog.

```
MAKE "Fred "dog
```

Now suppose you run this procedure:

```
TO try
LOCAL "Fred
MAKE "Fred "hound
SHOW "Fred
END
```

Before the procedure runs, SHOW :Fred gives dog as a response. Inside the procedure, a new variable, also called Fred, is created. It's the new, local variable that gets set to hound, and that's what will be printed by the SHOW command. When the procedure finishes, though, the local variable vanishes, and you'll see dog again if you run SHOW :Fred.

MAKE

```
MAKE name object
```

MAKE sets the value of the variable name to object. If the variable already exists, MAKE simply changes the value of the existing variable. If the variable doesn't exist, MAKE will create a new, global variable.

Snippet

```
MAKE "Colors [red orange yellow green blue violet]
```

NAME

```
NAME name object
```

NAME is another version of the MAKE command. The only difference is the value comes first, followed by the name of the variable.

Snippet

```
NAME [tall blonde] "Igor
```

NAMEP

```
NAMEP name
```

NAMEP returns TRUE if name is the name of a variable, and FALSE if it isn't.

Snippet

```
IF NOT NAMEP "Spot [MAKE "Spot "dog]
```

THING

```
THING variable
```

THING returns the value of name. It's completely equivalent to :variable. In fact, :variable is really just a shorter way of saying THING "variable.

Snippet

```
PR THING "Spot
```

Words and Lists

Numbers and Words

A lot of the commands in this section are used to pull apart words or put them together. In other languages, these would be called string manipulation commands. Logo shares a rather unusual ability with other artificial intelligence languages like LISP: A number is actually just a special kind of word. All of the commands that work on words will also work on numbers.

To see how this works, let's look at a couple of examples. Let's say you print a word, like this:

```
SHOW "010.0
```

Since there is a quote mark right before the characters, Logo treats this as a word, and prints `010.0`. If you take the quote mark off, though, Logo converts the value to a more efficient form, but you don't get exactly what you put in, either. When you try

```
SHOW 010.0
```

Logo responds with `10`.

When you use a number with a command that works on words, Logo does the same sort of conversion. For example,

```
FIRST "010.0
```

returns the word `0` (not the number!), while

```
FIRST 010.0
```

returns `1`, since the number is converted to 10 before `FIRST` gets a chance to work on the number.

Incidentally, you can also use words as input to calls that expect numbers; just make sure the string is a legal number before you use it as a parameter.

ASCII

`ASCII word`

`ASCII` returns the ASCII character code for the first letter in `word`.

Numbers are words, too. See "Numbers and Words," earlier in this chapter, for details.

Snippet

```
PR ASCII "a
```

BEFOREP

```
BEFOREP word1 word2
```

BEFOREP returns TRUE if word1 is before word2, and FALSE if it isn't.

The strings are compared character by character. One character is less than another if the ASCII character code for the character is less than the ASCII character code for the other character. For the most part, that means one character is less than the other if it comes first in alphabetical order, but all uppercase letters are less than lowercase letters.

If one word is shorter than the other, but the two words match up to the end of the shorter word, the shorter word is less than the longer one.

Numbers are words, too. See "Numbers and Words," earlier in this chapter, for details.

Tip You can use BEFOREP to sort words, but usually you want to sort words in dictionary order, ignoring the case. An easy way to sort the words is to use LOWERCASE or UPPERCASE to convert both words to the same case, then use BEFOREP to compare the words, like this:

```
BEFOREP LOWERCASE :word1 LOWERCASE
:word2
```

BUTFIRST

```
BUTFIRST object
BF object
```

BUTFIRST returns all of the object except the first element.

In the case of a list, BUTFIRST returns a list that has all of the elements from the input list except the first element. It is an error to use BUTFIRST on an empty list.

If object is a word, BUTFIRST returns the same word with the first character removed. If there is only one character, BUTFIRST returns a word with no characters. It is an error to give BUTFIRST a word with no characters as input.

Numbers are words, too. See "Numbers and Words," earlier in this chapter, for details.

Snippet

```
MAKE "Rest BUTFIRST :Cities
```

BUTLAST

```
BUTLAST object
BL object
```

BUTLAST returns all of the object except the last element.

In the case of a list, BUTLAST returns a list that has all of the elements from the input list except the last element. It is an error to use BUTLAST on an empty list.

If object is a word, BUTLAST returns the same word with the last character removed. If there is only one character, BUTLAST returns a word with no characters. It is an error to give BUTLAST a word with no characters as input.

Numbers are words, too. See "Numbers and Words," earlier in this chapter, for details.

Snippet

```
MAKE "Singular BUTLAST "words
```

CHAR

```
CHAR number
```

CHAR converts a number into the equivalent ASCII character. Compare this with the ASCII function, which converts characters to numbers.

Snippet

```
TO NextCh :ch
OP CHAR 1 + ASCII :ch
END
```

COUNT

```
COUNT object
```

COUNT returns the number of elements in object. For a list, this is the number of items in the list. For a word, this is the number of characters in the word.

Numbers are words, too. See "Numbers and Words," earlier in this chapter, for details.

Snippet

```
PR COUNT :Members
```

EMPTYP

```
EMPTYP object
```

EMPTYP returns TRUE if object is the empty list or a word with no characters, and FALSE if one of these conditions is not met.

Numbers are words, too. See "Numbers and Words," earlier in this chapter, for details.

Snippet

```
WHILE NOT EMPTYP :obj [TYPE LAST :obj MAKE "obj BL :obj]
```

EQUALP

```
EQUALP object1 object2
```

EQUALP returns TRUE if object1 and object2 are equal, and FALSE if they are not.

For numbers, the two objects are equal if they are the same number. If one of the objects is an expression, the expression is evaluated first, then compared to the other object.

For words, the two objects are equal if the words are identical. Letter case does matter, so "DAVE is not equal to "Dave.

For lists, the two objects are equal if the lists are equivalent. That means that each list has to contain exactly the same number of objects, and each object in the list has to be equal to the corresponding object in the other list.

Snippet

```
IF EQUALP "black :color [SETPC 0]
```

FIRST

```
FIRST object
```

FIRST returns the first element of object.

In the case of a list, FIRST returns the first element of the list. Unlike BUTFIRST and BUTLAST, FIRST doesn't return a list, unless, of course, the first element of object happens to be a list. It is an error to use FIRST on an empty list.

If object is a word, FIRST returns a word consisting of the first character in object. It is an error to give FIRST a word with no characters as input.

Numbers are words, too. See "Numbers and Words," earlier in this chapter, for details.

Snippet

```
PR FIRST [John Q. Doe]
```

FLOATP

```
FLOATP object
```

FLOATP returns TRUE if object is a floating-point number, and FALSE if it is anything else.

Snippet

```
IF NOT FLOATP :number [PR [I expected a price, like 4.35]]
```

FPUT

```
FPUT object list
```

FPUT places object at the start of list, and returns the new list.
LIST and LPUT can also be used to create or add to lists.

Snippet

```
FPUT "Fred [Sam Susan Karen]
```

INTEGERP

```
INTEGERP object
```

INTEGERP returns TRUE if object is an integer number, and FALSE if it is anything else.

Snippet

```
IF NOT INTEGERP :count [PR [I need a number!]]
```

ITEM

`ITEM integer object`

`ITEM` returns an element from `object`. `integer` tells `ITEM` which element to return. The items are numbered from 1 to the number of items in the object. `integer` must be greater than or equal to 1.

In the case of a list, `ITEM` returns an element of the list. Unlike `BUTFIRST` and `BUTLAST`, `ITEM` doesn't return a list, unless, of course, the element happens to be a list. If `integer` is larger than the number of elements in the list, `ITEM` will return an empty list.

If `object` is a word, `ITEM` returns a character from `object`. If `integer` is larger than the number of characters in the word, `ITEM` returns an empty string.

Numbers are words, too. See "Numbers and Words," earlier in this chapter, for details.

Snippet

```
MAKE "word ITEM 1 + :RANDOM 100 :HangmanWords
```

LAST

`LAST object`

`LAST` returns the last element of `object`.

In the case of a list, `LAST` returns the last element of the list. Unlike `BUTFIRST` and `BUTLAST`, `LAST` doesn't return a list, unless, of course, the last element of `object` happens to be a list. It is an error to use `LAST` on an empty list.

If `object` is a word, `LAST` returns a word consisting of the last character in `object`. It is an error to give `LAST` a word with no characters as input.

Numbers are words, too. See "Numbers and Words," earlier in this chapter, for details.

Snippet

```
TO Reverse :list
LOCAL "new
MAKE "new []
WHILE NOT EMPTYP :list [MAKE "new FPUT LAST :list :new MAKE
"list BUTLAST :list]
OUTPUT :new
END
```

LIST

```
LIST object1 object2
(LIST object1 object2 object3 ... )
```

LIST creates a list made up of the input objects. When you use the parenthesized form, you can give LIST any number of inputs, including a single input.

Compare LIST with SENTENCE, which also builds a list, but uses the *contents* of the objects instead of the objects themselves.

Snippet

```
MAKE "pets (LIST Fred Sam Psi)
```

LISTP

```
LISTP object
```

LISTP returns TRUE if object is a list, and FALSE if it is not a list.

Snippet

```
IF LISTP :parameter [ProcessList :parameter]
```

LOWERCASE

```
LOWERCASE word
```

LOWERCASE returns the input word after converting all of the uppercase letters in the word to the equivalent lowercase letters.

Numbers are words, too. See "Numbers and Words," earlier in this chapter, for details.

Snippet

```
IF EQUALP LOWERCASE :animal "dog [PR "Bark!]
```

LPUT

```
LPUT object list
```

`LPUT` places `object` at the end of `list`, and returns the new list.
`LIST` and `FPUT` can also be used to create or add to lists.

Snippet

```
MAKE "kids LPUT :who :kids
```

MEMBER

```
MEMBER object1 object2
```

`MEMBER` returns the part of `object2` that starts with `object1`.

If `object2` is a list, `MEMBER` checks each element of the list to see if it is equal to `object1`. If `MEMBER` finds a match, it returns a list that starts with the matching member, and contains all of the elements of `object2` from that element on. If there is no match, `MEMBER` returns the empty list.

If `object2` is a word, then `object1` must also be a word. In this case, `MEMBER` searches `object2` for a sequence of characters that matches `object1`. If it finds a match, `MEMBER` returns a word that starts with the first matching character from `object2`, and contains that character and all of the characters that follow it. If `MEMBER` does not find a match, it returns an empty word.

Snippet

```
IF NOT EMPTYP MEMBER :who :membership [PR [He is a member]]
```

MEMBERP

```
MEMBERP object1 object2
```

`MEMBERP` returns `TRUE` if `object1` is a member of `object2`, and `FALSE` if it is not a member of `object2`.

If `object2` is a list, `object1` is a member of `object2` if at least one of the elements of `object2` is equal to `object1`. The comparison is made exactly as if `EQUALP` were used to compare `object1` to each element of `object2`.

If `object2` is a word, then `object1` must also be a word. In this case, `object1` is a member of `object2` if `object1` appears somewhere in `object2`.

Snippet

```
IF NOT MEMBERP :who :membership [PR [He is a member]]
```

NUMBERP

```
NUMBERP object
```

`NUMBERP` returns `TRUE` if `object` is a number, and `FALSE` if it is not a number.

Snippet

```
IF NOT NUMBERP :value [PR [Please give me a number]]
```

PARSE

```
PARSE word
```

`PARSE` takes a word and breaks it up into a list, returning the list.

Snippet

```
MAKE "characters PARSE :who
```

SENTENCE

```
SENTENCE object1 object2
SE object1 object2
(SENTENCE object1 object2 object3 ... )
(SE object1 object2 object3 ... )
```

`SENTENCE` creates a list made up of the contents of the input objects. When you use the parenthesized form, you can give `SENTENCE` any number of inputs, including a single input.

If the objects are words or numbers, SENTENCE works just like its cousin, LIST. The difference between the two commands is how they handle objects that are lists. In this case, LIST puts the list itself into the output list, while SENTENCE puts the elements from the list into the output list. For example,

```
SHOW SENTENCE 1 [2 3 4]
```

prints [1 2 3 4], but

```
SHOW LIST 1 [2 3 4]
```

prints [1 [2 3 4]].

UPPERCASE

```
UPPERCASE word
```

UPPERCASE returns the input word after converting all of the lowercase letters in the word to the equivalent uppercase letters.

Numbers are words, too. See "Numbers and Words," earlier in this chapter, for details.

Snippet

```
TO Compare :word1 :word2
OP EQUALP UPPERCASE :word1 UPPERCASE :word2
END
```

WORD

```
WORD word1 word2
(WORD word1 word2 word3 ... )
```

WORD creates a word by combining all of the input words. It is an error if one of the inputs is a list.

Numbers are treated as words. See "Numbers and Words," earlier in this chapter, for details.

Snippet

```
TO Plural :word
OUTPUT WORD :word "s
END
```

WORDP

`WORDP object`

WORDP returns TRUE if `object` is a word, and FALSE if it is not a word.

Logo treats numbers as a special kind of word, so WORDP returns TRUE if object is a number, too. To see if an object is a word that is not a number, you have to make sure that WORDP returns TRUE and that NUMBERP returns FALSE.

Snippet

```
TO OnlyWordP :Object
IF NUMBERP :Object [OUTPUT "FALSE]
OUTPUT WORDP :Object
END
```

Property Lists

You can use property lists to organize information. A property list contains a series of names and properties. The names are used to find a property within the property list.

For example, you might want to keep track of the students in a class. One way to do this is with a group of property lists, one per student. Within the property list, you can keep track of various information about the students.

Here's how you can build a property list sing PPROP.

```
PPROP "Susan "who [Susan Patricia Westerfield]
PPROP "Susan "parents [Mike & Patty]
PPROP "Susan "phone [123 4567]
PPROP "Susan "transportation [bus route 1]
PPROP "Susan "likes [art science math]
PPROP "Susan "activities [swimming gymnastics]
MAKE "class LPUT :class "Susan
```

You can put all of the property list names in another list, then use the various list commands and property list commands to search and change the lists.

The obvious way to organize property lists is with the same names for each of the property lists you create, but you don't have to do it that way. In our example, you'd probably want to have a name for every student, but `activities` might be a special property you only create when you happen to know them.

GPROP

```
GPROP name property
```

GPROP gets a property from a property list. name is the name of the property list to check, while property is the property to look for. GPROP returns the value associated with property.

If there is no property list named name, or if the property list doesn't have a property named property, GPROP returns the empty list.

Snippet

```
SHOW GPROP "Susan "likes
```

PLIST

```
PLIST name
```

Returns the property list name. The property list is a list of property names and property values, with the names right after the value. For the example at the start of this section,

```
PLIST "Susan
```

would return a list that starts off

```
[name [Susan Patricia Westerfield] "parents [Mike & Patty] ...
```

PPROP

```
PPROP name property value
```

PPROP places a property and value in a property list. name is the property list PPROP will modify. property is the name of the property PPROP will set, while value is the value for the property.

If there isn't a property list called name, PPROP creates one.

If there is already a property in the property list with the name property, PPROP changes the value to value. If there isn't a property in the property list called property, PPROP adds a new property and value at the end of the property list.

Snippet

```
PPROP "Susan "activities [swimming gymnastics]
```

REMPROP

```
REMPROP name property
```

REMPROP removes property and it's value from the property list name. If property is the last property in the property list, the entire property list is removed from the workspace.

Snippet

```
REMPROP "Susan "likes
```

SETPLIST

```
SETPLIST name list
```

SETPLIST replaces all of the properties in the property list name with the properties in the property list list.

list is a list with an even number of elements. Counting from 1, the first element and all other odd numbered elements must be non-numeric words. These are the names for the properties in the property list. The entries in the list right after each property name is the property value for that name.

Snippet

```
SETPLIST "Frank PLIST "Defaults
```

Numbers and Arithmetic

One of the three forms for an object is a number. A Logo number is pretty much the same thing as what you think of as a number. Numbers can be whole numbers (integers in computer terms), like 1 or 47, or decimal numbers, like 4.5.

Logo uses two different ways to store numbers. Integers are whole numbers, like 4 or 1993. Integers have a definite range. Logo can handle integers from -2147483648 to 2147483647. If you type a whole number that is too big or too small to be an integer, Logo automatically converts the number to a floating-point number.

Floating-point numbers can have a fraction part, like 4.5 or 3.14159. They can also use scientific notation to represent very large or very small numbers, like 3e99. Floating-point numbers can range from 2.3e-208 to 1.7e+308.

Expressions

The obvious way to write a number object is just to type the number, but there's actually another way. Instead of writing the number, you can calculate it using Logo's math operations. Keep in mind that a number can literally be a value, like 4 or 3.14159, or it could be a number returned by some Logo function, or even a value returned by a function you write.

operator	use
-	When - appears right before a number, like -4, it changes the sign of the number.
+	The + operator is used between two numbers. It adds the two numbers together, returning the result.
-	When - appears after a number, Logo expects to find a second number right after the - operator. The right-hand number is subtracted from the left-hand number. For example, 3 - 7 returns -4.
*	The * operator is used between two numbers. It multiplies the two numbers, returning the result. For example, 4 * 5 is 20.
/	The / operator is used between two numbers. The number on the left is divided by the number on the right. For example, 5 / 4 returns 1.2, and 100 / 10 returns 10.

There are three other operations you can use in an expression that don't return a number; instead, they return either TRUE or FALSE. Each of these three comparison operators works on two numbers, comparing the number to the left of the operator to the number to the right of the operator.

operation	meaning
a < b	Returns TRUE of a is less than b, and FALSE if it is not.
a > b	Returns TRUE of a is greater than b, and FALSE if it is not.
a = b	Returns TRUE of a is equal to b, and FALSE if it is not.

Note Unlike the other math operators, = can be used on objects that are lists or words as well as on objects that are numbers. In fact,

```
object1 = object2
```

is just another way of writing

```
EQUALP object1 object2
```

The arithmetic operations use operator precedence to decide how calculations will be done. Operator precedence is just a fancy way of saying that some of the operations are done first, no matter what order they appear in. For example, when you write 1 + 2 * 3 in math class, you generally assume that the multiplication will be done first, then the addition, so the answer is 7.

Logo works the same way. Here's a table that shows the operator precedence from highest (the operations that will be done first) to lowest. In the case of a tie, the operations are done from left to right.

operation	notes
-	This is the unary subtraction, as in -4.
* /	
+ -	This is the subtraction operator, as in 2-4.
< > =	

You can use parenthesis to force Logo to do calculations in a particular order. For example, 1 + 2 * 3 is 7, but (1 + 2) * 3 is 9. Anything inside of a set of parenthesis is calculated completely before any operation from outside the parenthesis is done.

Anything that isn't in the table, like the built in math function `SIN` or one of your own functions, has a lower priority. Because of this rule

```
1 + SQRT 25 + 11
```

works like

```
1 + SQRT (25 + 11)
```

returning 7, not 17.

Special Rules for /

Most of the math operators are delimiters, too. Delimiters are special characters that Logo treats as the start of a new symbol. For example, if you type

```
:value*4
```

Logo knows that you want to multiply the contents of the variable `value` by 4. The reason is that `*` is a delimiter, so Logo treats it as the start of a new symbol, and treats the character that comes after it as the start of yet another symbol. The way Logo reads the line, you might as well have typed

```
:value * 4
```

In fact, that's what Logo will print when it lists a program that contains your original line.

The division operator, `/`, isn't a delimiter. Because of this, you need to make sure you put spaces around this character when you want it to be treated like a math operation. When you type

```
:value/4
```

Logo things you want to know the contents of a variable named `value/4`. When you type

```
:value / 4
```

Logo returns the contents of the variable value divided by 4.

There is a simple rule of thumb that will always keep you out of trouble: Always put a space on both sides of any math operator.

Special Rules for -

The character - is used for two completely different purposes. Because Logo is a list processing language, it's easy to get the purposes messed up. For example, is [2-5] a list with two elements, 2 and -5, or an expression that gives -3? That's very important when you're using a command like

```
SETPOS [20 -50]
```

Here are the rules Logo uses to decide what a - character really means:

- If a - character comes right before a number, and has any delimiter except a) character right before the - character (a space counts), the - character is treated like a negative sign for the number. For example,

  ```
  LAST [20 -50]
  ```

  is -50.

- If a - character comes right after a numeric expression, and the first rule doesn't apply, it's treated as a subtraction operator. For example, both of the - characters in

  ```
  SETPOS [XCOR - 10 YCOR-10]
  ```

  are subtractions.

- If the first rule doesn't apply, and the - character does not come after a numeric expression, it's a unary subtraction. For example,

  ```
  SETPOS [- XCOR YCOR]
  ```

  has a unary subtraction.

Trying to follow these rules literally isn't very easy. The easy way to stay out of trouble is to remember to always put spaces on both sides of a subtraction operation, and to always put a space

before, but not after, a minus sign. In addition, don't use minus signs in front of functions – use an expression, as in

```
SETPOS [0 - XCOR 0 - YCOR]
```

ABS

`ABS value`

`ABS` returns the absolute value of the argument. For example, `ABS -3` returns 3, while `ABS 10` returns 10.

ARCCOS

`ARCCOS value`

`ARCCOS` returns the angle whose cosine is `value`. `value` must be a number, or some combination of Logo statements that return a number.

Snippet

```
MAKE "angle ARCCOS :adjacent / :hypotenuse
```

ARCSIN

`ARCSIN value`

`ARCSIN` returns the angle whose sine is `value`. `value` must be a number, or some combination of Logo statements that return a number.

Snippet

```
MAKE "angle ARCSIN :opposite / :hypotenuse
```

ARCTAN

`ARCTAN value`

`ARCTAN` returns the angle whose tangent is `value`. `value` must be a number, or some combination of Logo statements that return a number. The angle that is returned will be in the range 270 to 359 or 0 to 90.

Snippet

```
MAKE "angle ARCTAN :opposite / :adjacent
```

ARCTAN2

`ARCTAN2 x y`

`ARCTAN2` returns the angle from a vertical line through the point [0 0] to [x y]. The angle that is returned will be greater than or equal to 0, and less than 360. The angle works just like turtle graphics, with 0 degrees being up, 90 degrees to the right, and so forth.

Snippet

```
MAKE "angle ARCTAN2 :adjacent :opposite
```

COS

`COS angle`

`COS` returns the cosine of the angle. `angle` must be a number, or some combination of Logo statements that return a number. Use degrees for the angle, just as you do with the Turtle Graphics commands.

▲ **Warning** If you use extremely large angles, the number you get back isn't very accurate. If you're doing a lot of math to calculate the angle, try and keep the result as close to the range of 0 to 359 degrees as possible. ▲

Snippet

```
MAKE "adjacent :hypotenuse * COS :angle
```

DIFFERENCE

```
DIFFERENCE value1 value2
```

DIFFERENCE returns value1 - value2. The inputs must both be numbers or statements that return numbers.

Snippet

```
PR DIFFERENCE :money :price
```

EXP

```
EXP number
```

EXP returns the exponent of number.

Snippet

```
TO PWR10 :value
OP EXP :value * LN 10.0
END
```

FLOAT

```
FLOAT number
```

FLOAT converts a numeric value to a floating-point value.

Tip The main reason for using FLOAT is to convert an integer into a floating-point number, which changes the way Logo does some math operations. For example, let's say you want to multiply two large numbers, like

```
SHOW 1000000 * 1000000
```

The answer is 1000000000000, but the largest integer Logo can handle is 2147483647. The largest floating-point number Logo can handle is a log bugger, though; it's about 1.7e308. By converting one of the integer inputs to a floating-point number, you tell Logo to do the multiplication using floating-

point math, and return the result as a floating-point number. When you type

```
SHOW 1000000 * FLOAT 1000000
```

Logo gives you 1.000000000000000e+12. The number is in scientific notation, but it is the right number. ◭

FORM

```
FORM number width digits
```

FORM formats a number, returning a word.
number is the number to format.
field is the field width. The word that FORM returns will have this many characters. If the number is so small that it doesn't need this many characters, blanks will be placed at the start of the word to force it to be width characters long. If width is too small, the number will be returned using the smallest number of characters possible. width must be a number from 1 to 128.
precision is the number of digits after the decimal point. It must be a number from 0 to 6. If you use 0, the number will be printed as an integer.

Some numbers are too large or too small to represent as integers or as numbers with a decimal point. If the absolute value of the number is smaller than 0.000001 or larger than 999999.0, the number will be printed using scientific notation.

Snippet

```
TO TYPE$ :amount
TYPE FORM :amount 1 2
END
```

INT

```
INT number
```

INT returns the integer part of a number. To get the integer part of a number, all of the digits to the right of the decimal place are removed.

Snippet

```
RIGHT INT :angle
```

INTQUOTIENT

```
INTQUOTIENT numerator denominator
```

INTQUOTIENT divides two integers, returning an integer result.
If denominator is zero, you will get a divide by zero error.
You can use real numbers for the inputs, but if you do, the inputs are converted to integers as if the INT function were used before the numbers are divided.

Snippet

```
(TYPE "You\ each\ get\  INTQUOTIENT :candy :kids "pieces.)
```

LN

```
LN number
```

LN returns the natural logarithm of number.

Snippet

```
PR EXP LN 3 + LN 4
```

POWER

```
POWER x y
```

POWER returns x raised to the power y. y must be greater than or equal to 0.

Snippet

```
PR :price * POWER 1.0 + :interest :months
```

PRODUCT

```
PRODUCT number1 number2
(PRODUCT number1 number2 number3 ... )
```

PRODUCT multiplies all of the input numbers, returning the result. If you use the second form, with parenthesis, it's possible to give PRODUCT a single number. In that case, PRODUCT returns the input number.

Snippet

```
(TYPE "The\ volume\ is\  (PRODUCT :length :width :height))
```

QUOTIENT

```
QUOTIENT numerator denominator
```

QUOTIENT divides two real numbers, returning a real number as the result. The inputs can be integers, but they will be converted to real numbers before the numbers are divided.

If denominator is zero, you will get a divide by zero error.

Snippet

```
MAKE "length QUOTIENT :area :height
```

RANDOM

```
RANDOM number
```

RANDOM returns a random integer that is greater than or equal to zero and less than number. See RERANDOM for a way to generate the same sequence of random numbers a second time.

Snippet

```
TO PICK :object
OP ITEM 1 + COUNT :object :object
END
```

REMAINDER

```
REMAINDER numerator denominator
```

REMAINDER returns the remainder from an integer division. The rules used to divide the numbers are the same as for INTQUOTIENT, but REMAINDER returns the remainder instead of the result of the division.

Snippet

```
MAKE "remaining REMAINDER :things :people
```

RERANDOM

```
RERANDOM
```

Computer generated random numbers aren't really random, of course. They are actually very specific numbers that are generated so they seem to be random. Random number generators start with a number called the seed. Logo uses the computer's clock to form a seed for its random number generator. The `RERANDOM` function will restart the random number generator, using the same value that was used for the very first call to `RANDOM`. When you do this, you will get the same sequence of random numbers again. Since Logo uses the clock to start the random number generator in the first place, though, the sequence will be different if you quit Logo and start again.

Snippet

```
PlayGame
RERANDOM
PlayGame
```

ROUND

```
ROUND number
```

`ROUND` converts real numbers to integers, returning the integer that is closest to the real number. Compare this with the `INTEGER` function, which returns the real number after removing the fraction part.

Snippet

```
TO ROUND$ :amount
OP QUOTIENT ROUND :amount * 100 100
END
```

SIN

```
SIN angle
```

`SIN` returns the sine of the angle. `angle` must be a number, or some combination of Logo statements that return a number. Use degrees for the angle, just as you do with the Turtle Graphics commands.

▲ **Warning** If you use extremely large angles, the number you get back isn't very accurate. If you're doing a lot of math to calculate

the angle, try and keep the result as close to the range of 0 to 359 degrees as possible. ▲

Snippet

```
MAKE "opposite :hypotenuse * COS :angle
```

SQRT

`SQRT value`

`SQRT` returns the square root of `value`. `value` must be a number or a statement that returns a number.

The square root is the number that, when multiplied by itself, gives `value`. You can't take the square root of a negative number; if you try, you'll get an error.

Snippet

```
MAKE "length SQRT X * X + Y * Y
```

SUM

```
SUM number1 number2
(SUM number1 number2 number3 ... )
```

`SUM` adds all of the input numbers, returning the result. If you use the second form, with parenthesis, it's possible to give `SUM` a single number. In that case, `SUM` returns the input number.

Snippet

```
PR (SUM 1 2 3 4 5)
```

TAN

`TAN angle`

`TAN` returns the tangent of the angle. `angle` must be a number, or some combination of Logo statements that return a number. Use degrees for the angle, just as you do with the Turtle Graphics commands.

▲ **Warning** If you use extremely large angles, the number you get back isn't very accurate. If you're doing a lot of math to calculate the angle, try and keep the result as close to the range of 0 to 359 degrees as possible. ▲

Snippet

```
MAKE "opposite :adjacent * TAN :angle
```

Flow of Control

CATCH

```
CATCH word list
```

CATCH runs list, just as if you used the RUN command. While the commands are running, though, you can use the THROW command. The THROW command jumps back to the CATCH command, stopping the execution of list. If CATCH is used in a procedure, the program will continue with the first statement after CATCH.

You can have more than one CATCH active at a time, so you need some way to tell the THROW command which CATCH command you want to go to. word is a label. The THROW command uses a label, too. Logo matches the THROW to the CATCH with the same label.

It's perfectly legal to THROW from some other procedure, which makes CATCH a perfect way to handle errors. For example, you can use a command like

```
CATCH "oops [DoButtonAction1]
```

to run a procedure called DoButtonAction1. From inside DoButtonAction1, or any procedure DoButtonAction1 calls, you can use a THROW command to gracefully stop the script if you find an error, like this:

```
IF errorDetected [THROW "oops]
```

There is one special CATCH label. If you use the label "ERROR, your CATCH statement will catch Logo errors, like divide by zero. If you catch the error, the script won't stop; it just jumps back to your CATCH statement and lets you handle the problem. You can use the ERROR statement to find out what error was flagged.

DOUNTIL

```
DOUNTIL list condition
```

DOUNTIL is a loop statement, sort of like REPEAT. DOUNTIL executes the statements in list as long as condition evaluates to FALSE. The instructions in list will always be evaluated at least once, since condition isn't checked until after list is executed.

Snippet

```
DOUNTIL [SETPOS MOUSE] NOT BUTTONP
```

ERROR

```
ERROR
```

ERROR is used to find out what error has occurred. When you use CATCH to catch errors, and Logo finds an error, it records the error number. ERROR returns a list; the first element of that list is the number of the last error caught by a CATCH statement. If there haven't been any errors caught, the error number will be zero.

Here's a list of the errors Logo can return. There are some gaps in the error numbers; that's to try and keep the error numbers in sync with other popular Logo implementations.

error	message
6	obj is a primitive
7	Can't find the label obj
9	obj is undefined
13	Can't divide by 0"
19	Too few items in name
20	Too many files are open
21	Can't find catch for obj
23	Out of space
25	obj is not TRUE or FALSE
29	Not enough inputs to name
30	Too many inputs to name
33	Can only do that in a procedure
34	Turtle out of bounds
35	I don't know how to name
38	You didn't say what to do with obj
41	name doesn't like obj as input
45	name is not open
57	Can't write to obj
58	Can't read from obj
59	A sound channel could not be started
60	Not enough memory to run Logo
61	Property lists must have an even number of elements

62	The directory `name` is not valid, or is not available
63	Disk I/O error
64	POFILE can only print text files
65	`name` wants a reader file
67	`name` could not find a window
68	`name` could not find the window `obj`

Snippet

```
IF NOT ERROR = 0 [PR [Abnormal termination]]
```

GO

```
GO label
```

`GO` jumps to the `LABEL` statement with the same label. `label` is a word.

There must be a `LABEL` statement with a matching label in the same procedure as the `GO` statement.

Snippet

```
GO "Fish
```

IF

```
IF condition trueList
IF condition trueList falseList
```

`IF` is used to pick between alternatives. `condition` is an expression that returns either `TRUE` or `FALSE`.

If `condition` is `TRUE`, `trueList` is executed, just as if you used the `RUN` command. `trueList` must be a list.

If `condition` is `FALSE`, and the second form of the `IF` statement is used, `falseList` is executed. Just as with `trueList`, `falseList` must be a list, and will be executed using the `RUN` command.

If `condition` is `FALSE` and the first form of the `IF` statement is used, the command does nothing.

The `IF` statement is a little unusual, since it can be either a command or a function. The `IF` statement returns whatever the list returns when it is executed. If the list doesn't return anything, or if you use the first form of the `IF` statement and condition is `FALSE`, if is a command. If one of the lists is executed and returns a value, `IF` returns the same value.

Snippet

```
PRINT IF :x < 0 [SQRT - :x] [SQRT :x]
```

IFFALSE

```
IFFALSE list
IFF list
```

If the last TEST statement was FALSE, IFFALSE runs list. If no TEST statement has been used in the current procedure, or if the last one returned TRUE, IFFALSE does nothing.

IFFALSE can only be used inside of a procedure.

list must not return a result. IFFALSE is a command, and does not return a result.

Snippet

```
IFFALSE [PR [Sorry\, that's not right.]
```

IFTRUE

```
IFTRUE list
IFT list
```

If the last TEST statement was TRUE, IFTRUE runs list. If no TEST statement has been used in the current procedure, or if the last one returned FALSE, IFTRUE does nothing.

IFTRUE can only be used inside of a procedure.

list must not return a result. IFTRUE is a command, and does not return a result.

Snippet

```
IFTRUE [PR [That's right!]]
```

LABEL

```
LABEL label
```

LABEL is used to create destinations for GO statements. If your program happens to execute a LABEL statement, it just gets skipped.

label must be a word, and it has to be a word constant. For example, you cannot call a function that returns the word.

Snippet

```
LABEL "Fish
```

OUTPUT

```
OUTPUT object
OP object
```

OUTPUT returns from the current procedure, returning a value in the process. When you leave a procedure with OUTPUT, your procedure becomes a function; the result is object.

OUTPUT can only be used inside of a procedure.

See STOP for a way to return from a procedure without returning a value.

Snippet

```
TO ! :value
LOCAL "x
MAKE "x 1
WHILE :value > 1 [MAKE "x :x * :value MAKE "value :value - 1]
OP :x
END
```

REPEAT

```
REPEAT count list
```

REPEAT runs list count times.

Snippet

```
REPEAT 4 [FD 20 RT 90]
```

RUN

```
RUN list
```

RUN runs list just as if it were typed from the input line. If list is an operation, RUN writes the result to the screen.

The RUN command gives Logo one of it's most powerful features: with some care, you can create a Logo program from inside your own Logo program by forming a list. Once the list has been created, you can run the list from inside your program.

Snippet

```
TO Apply :command :list
IF EMPTYP :list [STOP]
RUN LIST :command FIRST :list
Apply :command BUTFIRST :list
END
```

STOP

STOP

STOP returns from the current procedure. When you leave a procedure with STOP, your procedure acts like a command.

You don't need to use STOP to return from a procedure if you are already an the end of the procedure. The procedure stops automatically when it runs out of statements.

STOP can only be used inside of a procedure.

See OUTPUT for a way to return a value when you return from a procedure.

Snippet

```
IF :done [STOP]
```

TEST

TEST condition

TEST evaluates condition, setting a flag that is used later by IFTRUE and IFFALSE statements. Each procedure has its own test flag, so using TEST in one procedure doesn't effect the way other procedures work.

condition must return either TRUE or FALSE.

Snippet

```
TEST EQUALP :reply :answer
```

THROW

```
THROW word
```

THROW jumps back to the CATCH statement whose label is word. It is an error if there is no CATCH statement to catch this throw.

See the description of the CATCH command for details on how THROW is used.

Snippet

```
THROW "Exit
```

TOPLEVEL

```
TOPLEVEL
```

TOPLEVEL stops a program. If it is executed while a button script is running, the script stops. If it is executed while you are using the console, whatever is running is stopped and you can type a new command.

Snippet

```
IF ErrorFound [TOPLEVEL]
```

WAIT

```
WAIT value
```

WAIT waits for value 1/60ths of a second. You can use WAIT to pause in a program. While WAIT isn't accurate enough to use as a clock, it isn't effected by the machine you are using – it will wait just as long on an unaccelerated Apple IIGS as on one with an accelerator card.

Snippet

```
WAIT 60
```

WHILE

```
WHILE condition list
```

WHILE is a loop statement, sort of like REPEAT. The difference is that REPEAT executes a list for a specific number of times, but WHILE executes a list as long as condition is TRUE. As soon as condition is not TRUE, WHILE stops.

What actually happens is this: WHILE starts by evaluating condition. If the result is TRUE, the statements in list are executed, and the WHILE statement starts over. If condition is not TRUE, WHILE stops immediately and moves on to the next statement.

Since the condition is checked right away, a while loop doesn't necessarily execute the statements in list at all. If condition is not TRUE the first time it is evaluated, all WHILE does is make sure list is, in fact, a list.

Snippet

```
WHILE BUTTONP [SETPOS MOUSE]
```

Logical Operators

AND

```
AND object1 object2
(AND object1 object2 object3 ...)
```

AND returns TRUE if all of the input objects are TRUE, and FALSE in any one of the input objects is FALSE. All of the inputs must be TRUE or FALSE.

Snippet

```
TO Range :start :value :end
OP :start < :value AND :value < :start
END
```

NOT

```
NOT object
```

NOT returns TRUE if object is FALSE, and FALSE if object is TRUE. object must be TRUE or FALSE.

Snippet

```
PR NOT :a < :b
```

OR

```
OR object1 object2
(OR object1 object2 object3 ...)
```

OR returns TRUE if at least one of the input objects are TRUE, and FALSE if all of the input objects are FALSE. All of the inputs must be TRUE or FALSE.

Snippet

```
PR :a < :b OR :a = :b
```

Input and Output

The input and output commands are used to communicate with the outside world.

BUTTONP

```
BUTTONP
```

BUTTONP returns TRUE if the mouse button is down, and FALSE if it is not being pressed.

Snippet

```
WHILE BUTTONP [SETPOS MOUSE]
```

KEYP

KEYP

KEYP returns TRUE if a key is available from the current input device (usually the keyboard), and FALSE if not.

Snippet

```
IF KEYP [DoCommand]
```

MOUSE

MOUSE

MOUSE returns a two-element list. The first element is the horizontal mouse position, using Turtle Graphics coordinates. The second element of the list is the vertical mouse position. If there is no active window for the turtle, MOUSE returns [0 0].

Snippet

```
WHILE BUTTONP [SETPOS MOUSE]
```

PRINT

```
PRINT object
(PRINT object1 object2 object3 ...)
PR object
(PR object1 object2 object3 ...)
```

PRINT prints the input objects, printing each one on a separate line. It follows the last object with a carriage return, too, so anything you print with another command will start on a fresh line.

PRINT does not print brackets if the object is a list.

Note SHOW and TYPE are similar to SHOW, but work in slightly different ways.

Snippet

```
PRINT [Hello\, world.]
```

READCHAR

```
READCHAR
RC
```

READCHAR returns a key from the current input device, usually the keyboard. If a key has not been pressed, READCHAR waits for a keypress before returning. The key that is pressed is not printed.

Tip You can use KEYP to see if a key is available.

Snippet

```
MAKE "ch READCHAR
```

READCHARS

```
READCHARS number
RCS number
```

READCHARS reads number characters from the keyboard. It returns all of the characters in a word.

Snippet

```
MAKE "zipcode READCHARS 5
```

READLIST

```
READLIST
RL
```

READLIST reads a line from the input device (usually the keyboard) and converts the input into a list. READLIST returns the list. It's equivalent to PARSE READWORD.

Snippet

```
MAKE "address READLIST
```

READWORD

READWORD

READWORD reads a line from the current input file, returning the line it reads as a word. Normally that's the keyboard, but you can change where the lines are read from using SETREAD.

Snippet

```
MAKE "answer READWORD
```

SHOW

SHOW object

SHOW prints object, followed by a carriage return. If object is a list, SHOW will print brackets around the list.

You can use this command to write to a file; see SETWRITE.

Note PRINT and TYPE are similar to SHOW, but work in slightly different ways.

Snippet

```
SHOW :value
```

TEXTIO

TEXTIO

TEXTIO tells Logo to write to the current text window. That's where it normally writes anyway, so you won't need this command unless you've already used TURTLEIO to tell Logo to write to the current turtle port.

Snippet

```
TEXTIO
```

TOOT

```
TOOT frequency duration
```

`TOOT` plays a note.
The note will last for `duration` 1/60ths of a second.
`frequency` is the MIDI frequency for the note. It must be a number from 1 to 127.

	A	A#	B	C	C#	D	D#	E	F	F#	G	G#
Octave 1					1	2	3	4	5	6	7	8
Octave 2	9	10	11	12	13	14	15	16	17	18	19	20
Octave 3	21	22	23	24	25	26	27	28	29	30	31	32
Octave 4	33	34	35	36	37	38	39	40	41	42	43	44
Octave 5	45	46	47	48	49	50	51	52	53	54	55	56
Octave 6	57	58	59	60	61	62	63	64	65	66	67	68
Octave 7	69	70	71	72	73	74	75	76	77	78	79	80
Octave 8	81	82	83	84	85	86	87	88	89	90	91	92
Octave 9	93	94	95	96	97	98	99	100	101	102	103	104
Octave 10	105	106	107	108	109	110	111	112	113	114	115	116
Octave 11	117	118	119	120	121	122	123	124	125	126	127	

MIDI `frequency` Values and the Musical Scale

Snippet

```
TOOT 60 60
```

TURTLEIO

```
TURTLEIO
```

`TURTLEIO` tells Logo to write to the current turtle window. Once you use this command, Logo draws text in the turtle window instead of writing it in the text window.

Logo switches back to the text window as soon as it finishes running the command you type.

The obvious use of `TURTLEIO` is to write text onto a drawing, but there are other benefits, too. When you draw text in the turtle window, you get a lot more control over the way the text is drawn. You can change the font, the size of the letters, the style used to write the letters, the color, and even draw letters in 3D. See "Fonts" for more information about drawing text and the commands you use to control it.

△ **Important** The fact that Logo switches back to the text screen when it is waiting for you to type a command can cause some surprises. The biggest is that typing

```
TURTLEIO
PR "Hello
```

doesn't write "Hello" to the turtle window; it is written in the text window. The reason, of course, is that Logo switched back to the text screen after you typed TURTLEIO, while it was waiting for you to type the next command. To write to the turtle window, either put both commands in a procedure, so they will be executed before Logo needs more input, or type them on one line, like this:

```
turtleio  pr "Hello △
```

TURTLEIO changes the way Logo reads text, too. Once you switch to graphics input and output, reading a line with READLIST or READWORD doesn't wait for you to type something in a text window. Instead, Logo brings up a dialog box with a line edit box where you can type text and an OK button. You type whatever you want in the dialog, then press OK – the input command gets the characters, just as if you had typed them from a text window.

The keyboard input commands (READCHAR and READCHARS) are not effected by this command. They still read characters directly from the keyboard, never displaying the character they read.

TYPE

```
TYPE object
(TYPE object1 object2 object3)
```

TYPE prints the objects. It does not print brackets if the object is a list, and it does not print a carriage return after printing the object. If you use TYPE to print more than one object, they will be jammed together on a single line.

Note SHOW and TYPE are similar to SHOW, but work in slightly different ways.

Snippet

```
TYPE 1 + 3
```

Disk Commands

File Names

Several of the commands in this section use file names. File names are words that describe the name of a file and where it is located on a disk. There are a lot of different ways to give a file name, and some seem a little complicated the first time you see them. We'll cover two of the most common, and easiest to understand, kinds of file names here. You can find more detailed information about file names from a variety of sources, including the technical reference material Apple publishes for programmers.

GS/OS is a very flexible operating system that supports more than one disk format. Each of the different disk formats have slightly different rules for what is allowed and what is not allowed in a file name. We'll describe the most common disk format, ProDOS, here. The rules for ProDOS names will work on all of the other commonly used disk formats, too.

Every file on your disk has a name. The file name is what you see under the icon for a file when you look at the file from the Finder. The file name is used to identify a file uniquely, just like your name identifies you. You can use up to 15 characters in a file name. File names must start with a letter. After this starting letter, you can use letters, numbers, or the period.

As you know, you can use either uppercase or lowercase letters in a file name, but what you might not know is that, while your computer remembers the case of the letters you use, it doesn't treat them as different in any way. In programming terms, file names are case insensitive.

▲ **Warning** The fact that file names are case insensitive is very important. For example, let's say you save your workspace with this command:

```
SAVE "mystuff
```

If you go to another stack, and save its workspace with this command:

```
SAVE "MYSTUFF
```

you might think that you were saving the information to a new file, but that isn't the case. The names may look different on the disk, but your computer treats them as the same name, wiping out your original file. ▲

If there are two people named Susan in a classroom, you can usually tell them apart by using their last name. You can have two files on your computer with the same name, too, as long as they are in different folders. To tell the computer which file you mean, you can use a full path name, which works sort of like a person's last name.

The full path name starts with the name of the disk the file is on. Next comes the folder the file is in (if it's in one). The disk name, folder names, and the name of the file are all separated with colon (':') characters. You also use a colon before the name of the disk itself.

For example, to save a file called `mystuff` to a disk named `mydisk`, you use the command

```
SAVE ":mydisk:mystuff
```

To put a different file called `mystuff` on the same disk, but this time inside of a folder named `myfolder`, use the command

```
SAVE ":mydisk:myfolder:mystuff
```

If the file goes in a folder that is inside of another folder, just tack all of the folder names together, separated by colons, in the same order you would open them from the Finder.

ALLOPEN

`ALLOPEN`

`ALLOPEN` returns a list of all of the open files. If there are no open files, `ALLOPEN` returns the empty list.

Snippet

```
ALLOPEN
```

CATALOG

`CATALOG`

`CATALOG` lists all of the files in the current folder. To the left of each file name are two fields; the number of blocks the file use on the disk, and the type of the file.

```
407     ---         Logo3D.Sys16
75      ---         Installer
1       DIR         Scripts
1       DIR         Samples
1       ---         Finder.Root
1       ---         Finder.Data

Blocks Used: 553   Blocks Free: 1047
```

The file type is one of six things:

file type	meaning
DIR	A directory (folder).
TXT	A text file you can open and read with the Open command from the File menu.
ANI	An animation file, or movie. You can open and play these movies from Logo.
PNT	A picture file that you can open; it will become a turtle window.
LOG	A Logo program. You can open this file with the Open command to edit it, or execute the file as a program.
---	Any file Logo can't use is shown with dashes for the file type.

Catalog also shows the total number of blocks on the disk and the number of free blocks left on the disk. On most disks, a block is 512 bytes, but due to the way files are stored on disk, the number of blocks the file uses doesn't always have an exact relationship to the number of bytes in the file.

Snippet

```
CATALOG
```

CLOSE

```
CLOSE name
```

`CLOSE` closes a file that was opened with `OPEN`. It is an error to close a file that is not open.

Snippet

```
CLOSE "myfile
```

CLOSEALL

```
CLOSEALL
```

`CLOSEALL` closes all files that are open. You can use `CLOSEALL` even if no files are open.

Snippet

```
CLOSEALL
```

CREATEFOLDER

```
CREATEFOLDER name
```

CREATEFOLDER makes a new directory (folder).

Snippet

```
CREATEFOLDER "myfolder
```

DIR

```
DIR
```

DIR returns a list of the names of all of the files in the current folder.
See SETPREFIX for a way to change the current folder.

Snippet

```
PR DIR
```

ERASEFILE

```
ERASEFILE file
```

ERASEFILE removes a file from a disk. Once a file is erased, you can't recover it. The space used by the file is available for other files.

Some programs can lock files, protecting them from ERASEFILE.

You can erase a directory (folder) with ERASEFILE, but only if all of the files inside of the directory have already been erased.

Snippet

```
ERASEFILE "myfile
```

FILELEN

```
FILELEN name
```

FILELEN returns the length of a file in bytes. (Loosely speaking, one byte is one character.) The file must be open before you can use FILELEN to get the length.

Snippet

```
FILELEN "myfile
```

FILEP

```
FILEP file
```

FILEP checks to see if a file exists. If the file exists, FILEP returns TRUE; if the file does not exist, FILEP returns FALSE.

Snippet

```
IF FILEP "temp [ERASEFILE "temp]
```

LOAD

```
LOAD file
```

LOAD loads a file from your disk. The file itself must be a text file. Logo reads and executes the contents of the file just as if you had typed the file from a text window.

The most common way to use LOAD is to save and load files and variables from your workspace. For example, you could save your workspace, then load the procedures again the next time you use Logo. Once you have saved a file, you can even edit it with a text editor and load the results – or create a file of procedures and variable from scratch.

See "File Names," earlier in this chapter, for information about legal file names.

Snippet

```
LOAD "Cubes
```

ONLINE

```
ONLINE
```

ONLINE shows the name of all disks that Logo can use.

You can see what files are on the disk by using SETPREFIX to set the current folder to the disk, then using either DIR or CATALOG.

Snippet

```
ONLINE
```

OPEN

```
OPEN name
```

`OPEN` opens a file for reading and writing. If the file doesn't already exist, `OPEN` creates a new, empty file.

You must open a file before using the file input and file output procedures described in this section. In general, you will follow `OPEN` with a call to `SETREAD` or `SETWRITE`.

Snippet

```
OPEN "myfile
```

POFILE

```
POFILE name
```

`POFILE` prints a file. The file must be a text file. (Text files have a GS/OS file type of TXT or SRC. Logo saves programs using SRC files.)

Snippet

```
POFILE "myfile
```

PREFIX

```
PREFIX
```

`PREFIX` returns the default prefix.

When you use a partial path name of file name, Logo forms a full path name by attaching your partial file name to the default prefix. For example, if the prefix is :MYDISK:MYFOLDER:, and you erase the file MYFILE with the command

```
ERASEFILE myfile
```

Logo forms the full path name :MYDISK:MYFOLDER:MYFILE, and deletes this file.

You can set the default prefix with the `SETPREFIX` command.

Snippet

```
MAKE "old PREFIX
```

READER

```
READER
```

READER returns the name of the file that is open for reading. If there isn't an open reader file, READER returns the empty list.
You can open a file for reading with SETREAD.

Snippet

```
PR READER
```

READPOS

```
READPOS
```

READPOS returns the number of bytes that have already been read from a file.

Snippet

```
SETREADPOS READPOS + 10
```

RENAME

```
RENAME oldname newname
```

RENAME changes the name of a file, folder or disk. The name is changed from oldname to newname.

Snippet

```
RENAME "myfile "yourfile
```

SAVE

```
SAVE file
```

SAVE saves a file to your disk. It writes all of the variables, property lists and procedures to a disk file just as if you had used the POALL command and captured the results in the file – which, as a matter of fact, is actually what happens. The file is a text file with no owner, which means you can load it with pretty much any program that can read a text file.

See "file Names," earlier in this chapter, for information about legal file names.

Snippet

```
SAVE "Cubes
```

SETPREFIX

```
SETPREFIX prefix
```

SETPREFIX changes the default prefix to prefix.

When you use a partial path name of file name, Logo forms a full path name by attaching your partial file name to the default prefix. For example, if the prefix is :MYDISK:MYFOLDER:, and you erase the file MYFILE with the command

```
ERASEFILE myfile
```

Logo forms the full path name :MYDISK:MYFOLDER:MYFILE, and deletes this file.

When you use SETPREFIX to change the prefix, you can use either a full path name (one that starts with a colon and the name of a disk) or a partial path name. If you use a partial path name, it is added to the current prefix to form the new default prefix. This is a good way to burrow deeper into a directory structure. For example, let's say you put in the disk :MYDISK:, and set the default prefix to the new disk and catalog the disk with the commands

```
SETPREFIX ":mydisk:
CATALOG
```

If you see a folder called MYFOLDER, you could use a full path name to set the default prefix to MYFOLDER with the command

```
SETPREFIX ":mydisk:myfolder:
```

but you could save some typing by using a partial path name:

```
SETPREFIX "myfolder
```

SETREAD

```
SETREAD name
```

SETREAD makes a file the reader. You must open a file before you can use SETREAD to make the file the reader.

Once the file is the reader, READWORD, READCHAR, READCHARS and READLIST will all read from the file.

To stop reading from the reader file and switch back to reading the keyboard, either close the file or set the reader to the empty list, like this:

```
SETREAD []
```

You can also set the reader to a different open file. When you switch back, input will pick up from the spot it was at when you switched the reader.

SETREADPOS

```
SETREADPOS position
```

SETREADPOS sets the position in the current reader file. You can use SETREADPOS to jump around in an open file, reading first one part, then another.

It is an error if the position is past the end of the file, or if there is no open reader.

Snippet

```
SETREADPOS :record * :length
```

SETWRITE

```
SETWRITE name
```

SETWRITE makes a file the writer. You must open a file before you can use SETWRITE to make the file the writer.

Once the file is the writer, SHOW, PRINT and TYPE will all write to the file.

To stop writing to the writer file and switch back to writing to the text window, either close the file or set the writer to the empty list, like this:

```
SETWRITE []
```

You can also set the writer to a different open file. When you switch back, you'll start writing at the spot it you were writing to before you switched the writer.

SETWRITEPOS

```
SETWRITEPOS position
```

SETWRITEPOS sets the position in the current writer file. You can use SETWRITEPOS to jump around in an open file, writing to first one part, then another. This is one way to create a random access database, which lets you write a record that has been changed, even if it is in the middle of a file.

It is an error if the position is past the end of the file, or if there is no open writer.

Snippet

```
SETWRITEPOS :record * :length
```

WRITEPOS

```
WRITEPOS
```

WRITEPOS returns the number of bytes that have already been written to a file. This isn't necessarily the total number of bytes in the file, since SETWRITEPOS can be used to jump around in the file. To find the total number of bytes in the file, use FILELEN.

Snippet

```
MAKE "record WRITEPOS / :length
```

WRITER

```
WRITER
```

WRITER returns the name of the file that is open for writing. If there isn't an open writer file, WRITER returns the empty list.

You can open a file for reading with SETWRITE.

Snippet

```
PR WRITER
```

Turtle Graphics

Turtle Positions

Turtle graphics uses a standard Cartesian coordinate system. `[0 0]` is at the center of a window. These windows are as large as one full display screen, although you can only see a portion of the full screen in the window. (The scroll bars can be used to show different parts of the screen.) The first coordinate, X, increases as you move to the right, and decreases as you move to the left. The second coordinate, Y, increases as you move up, and decreases as you move down. Unless you use a scrunch factor, one unit is the same as one screen pixel in 320 mode. In 640 mode, one unit is two pixels wide.

You can also use a third dimension, Z. The Z coordinate is 0 in the plane of the display screen, and increases as you move from the screen towards your eye. Negative Z coordinates move into the computer display. See "3D Turtle", a little later in this chapter, for more information about the 3D display systems.

Turtle Heading

The turtle is always pointed in some direction, and when you use `FORWARD` or `BACK` to move the turtle, it moves off in the direction it is pointed. The turtle uses compass headings for direction. Zero degrees is up, pointed towards the top of the screen. The angles move clockwise, so 90 degrees points to the right, 180 degrees points down, and 270 degrees points left.

If you give the turtle a heading that isn't between 0 and 360 degrees, the heading is adjusted. For example, if you use `SETHEADING` to set the heading to -90 degrees, then look at the heading with `HEADING`, you'll get 270. For the most part, you can just let the turtle adjust the angles for itself, but you do need to be careful if you're working with very large numbers. The larger the number, the less accurate it will be when it's adjusted.

When you use the 3D turtle, the normal 2D heading works the same way, but it's called a longitude angle instead of a heading. A second angle, latitude, points the turtle out of the screen or into the screen; and a third angle, roll, rotates the turtle back and forth.

With the second angle, the turtle heading works like longitude and latitude angles on a globe, with the turtle in the middle of the globe, the equator lying on the screen, and the north pole towards you. Zero degrees latitude keeps the turtle in the screen plane, where it will work just like a 2D turtle. Ninety degrees points the turtle straight up, towards your eye and towards the north pole. Minus 90 degrees points the turtle straight down, into the screen and towards the south pole.

The third angle rolls the turtle back and forth. Think about how `LEFT` and `RIGHT` work with the normal, two-dimensional turtle. When you say `RIGHT 90`, the turtle doesn't point to the right, it moves 90 degrees further to the right from whatever its current heading happens to be. Floating in space, the turtle will still rotate to its own right. Now think about the turtle sitting on the screen, just like a normal 2D Logo turtle, and pointed straight up. `RIGHT 90` works just

like it always did. Now imagine that the turtle is starting out pointed straight up again, but this time you say `ROLLLEFT 90`, then `RIGHT 90`. This time the turtle is pointed straight up.

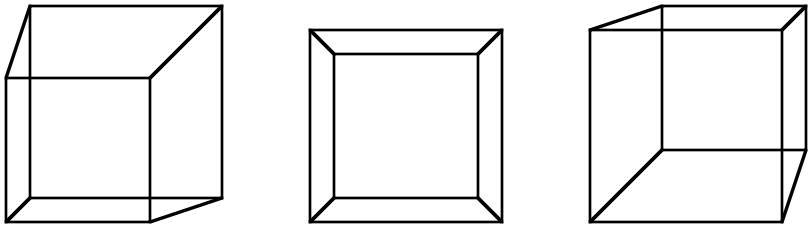
A lot of folks used to laugh at fighter pilots, who always talk with their hands, but you should try the same trick. If you're having any trouble seeing what is happening with the headings, read the last paragraph again, but this time, rotate your hand.

3D Turtle

The last two sections mentioned that 3D Logo supports a 3D turtle, rather than the standard 2D turtle you find in most Logos. "Turtle Positions" explained that the Z coordinate is zero at the screen level, with positive values moving out of the screen and negative values moving into the screen. "Turtle Headings" explained that the heading works like latitude and longitude, with the turtle at the middle of the globe, and a roll angle.

Of course, as you can easily see, your computer can't really draw in three dimensions... or can it? Actually, 3D Logo lets you draw in three dimensions two different ways.

You've seen three-dimensional objects displayed in two dimensions all your life, and the result can be very realistic. Television screens, pictures in books, photographs, and some engineering drawings are just a few examples of effective ways to put a three dimensional image on a flat display. Basically, these three dimensional display systems show you an image as if you closed one eye. In this book, we'll call that a projection display. If you draw a cube using the projection display, the back edges will look a little smaller than the front edges, like this:



Projections of a Cube

Graphics books call this a wire frame drawing. It's sort of like a three-dimensional version of a simple line drawing, with each edge represented by a wire instead of a line. With a wire frame drawing, you see all of the outlines, even the ones that would be invisible if the object were solid.

This sort of projection is about as good as it gets with a standard, flat display. You can improve on the drawing to get a more realistic picture, of course, but it's still a flat projection of a three dimensional scene, not a true three dimensional display. To get three dimensions, each of your eyes has to see a slightly different picture – the left eye needs to see a scene from an point about two to two and a half inches away from the scene your right eye sees. (The exact distance depends on the spacing between your eyes. For adults, it's close to two and a half inches; for second grade kids, it's pretty close to two inches.) Your brain uses the two different images to figure out how far away the object you are looking at happens to be.

One way to trick your brain into thinking it is seeing a three dimensional object is to show it two different pictures, one in each eye, each drawn from a slightly different viewpoint. That's exactly what 3D Logo does. When you use the stereoscopic display system, moving the turtle actually draws two lines. When you draw a complete picture, you get two complete drawings, one for the left eye and one for the right eye. The picture your left eye should see is drawn in red, and the picture for the right eye is drawn in blue, and both are drawn against a black background. If you're not wearing the special 3D glasses, the result is, well, strange. In really doesn't look like much. When you put the glasses on, though, the effect is very realistic. In fact, it's so realistic that one of my second grade beta testers actually reached out to touch the drawing! She giggled when she grabbed air about four inches in front of the computer screen.

Use the CLEARSCREEN3D command to turn on the stereoscopic drawing mode. CLEARSCREEN3D changes the background to black, clears the screen, and puts the turtle in the middle of the screen pointed up. All of the turtle graphics commands except FILL and DOTP will start working in three dimensions. You can use SHOWTURTLE3D to turn on the 3D turtle without clearing the screen to black, but the effect is still a lot more realistic when you draw on a black background.

Some of the normal turtle graphics commands work a little differently, since they have a new dimension and two new angles. The commands that work differently are DOT, HEADING, POS and SETPOS. The other commands support the stereoscopic display, but you use them the same way you do for a two dimensional turtle. For example, FORWARD still moves forward, but forward might be straight out of the screen with a three dimensional turtle.

You can also draw wire frame drawings using the projection display system. There are a couple of reasons you might want to use the projection display instead of the stereoscopic display. First, the stereoscopic display only works in black and white, since it is using color to split the screen display between your two eyes. For a color drawing, you should stick with the projection display. The other reason to use the projection display is when you don't want to use the 3D glasses, or when some people who are looking over your shoulder don't have the glasses.

The projection display turns on automatically any time you use a command with a Z coordinate or set a heading that isn't in the screen plane, and CLEARSCREEN3D or SHOWTURTLE3D haven't already been used to turn on the stereoscopic display. The commands that can turn on the projection screen are DOT, ROTATEIN, ROTATEOUT, ROLLLEFT, ROLLRIGHT, SEATHEADING3D, SETPOS and SETZ.

Turtle Colors in 2D

Several of the turtle graphics commands support color. These commands use a number to pick a color, or to tell you what color something is. The color numbers depend on whether you are using 320 mode graphics or 640 mode graphics. It's also possible to change the colors using desk accessories.

In 320 mode, the colors are:

number	color	number	color
0	black	8	beige
1	dark gray	9	yellow
2	brown	10	green
3	purple	11	light blue
4	blue	12	lilac
5	dark green	13	periwinkle blue
6	orange	14	light gray
7	red	15	white

In 640 mode, there are actually only four pure colors, but they are combined in a special way when pixels of differing colors appear next to each other. This effect, called dithering, gives a total of 16 colors – but there are some limits.

First, if a line is too thin, you may not get the color you want. Normally, the Logo pen is 2 pixels wide in 640 mode. That will give you all 16 colors, but if you have too many lines too close together, you may end up with fringing effects that spoil the colors. To avoid these fringing effects, stick to the pure colors – color numbers 0, 3, 10 and 15.

Assuming the objects you are drawing are not too narrow, the colors in 640 mode are:

number	color	number	color
0	black	8	green
1	dark blue	9	blue green
2	olive	10	spring green
3	gray	11	light green
4	red	12	dark gray
5	purple	13	periwinkle blue
6	orange	14	yellow
7	peach	15	white

Turtle Colors in 3D

When you are using the stereoscopic display system, you don't have the same range of colors you get with the 2D turtle. Instead, the colors are mapped into gray scales.

In both 320 mode and 640 mode, the background defaults to black, which is color 0.

In 320 mode, you have three additional gray colors, 1, 2 and 3. Three is the brightest color, and the one you'll normally want to use.

In 640 mode, there are only two colors. They are 0 (black) and 1 (white).

It's not an error to use a color number for a color that doesn't exist. When you do, Logo automatically maps the color number you give into one that's available.

BACK

```
BACK distance
BK distance
```

The turtle moves back `distance` pixels. If the pen is down, a line is drawn from the old turtle location to the new turtle location.

Snippet

```
BACK 40
```

BACKGROUND

```
BACKGROUND
BG
```

Returns a number from 0 to 15; this is the current background color.
`SETBG` sets the background color.
See "Turtle Colors," earlier in this chapter, for a list of the turtle colors.

Snippet

```
MAKE "oldcolor BACKGROUND
```

CLEAN

```
CLEAN
```

Erases the turtle window, painting the entire window with the background color. The turtle's position and heading do not change.

Snippet

```
CLEAN
```

CLEARSCREEN

```
CLEARSCREEN
CS
```

Clears the screen, puts the turtle at [0 0], and sets the turtle heading to 0.
See "Turtle Positions", earlier in this chapter, for details on the turtle coordinate system.

Snippet

```
CLEARSCREEN
```

CLEARSCREEN3D

```
CLEARSCREEN3D
CS3D
```

Clears the screen, puts the turtle at [0 0 0], and sets the turtle heading to [0 0 0]. The background color is set to black before clearing the screen. Once this command is used, all turtle drawing commands will create stereo images that appear as true 3D images when you look at the screen with the special 3D glasses.
See "3D Turtle", earlier in this chapter, for details on the 3D turtle.

Note `CLEARSCREEN3D` turns on the stereoscopic 3D display system. Anything you draw using this display model will look crummy without 3D glasses, but will display as true 3D with the glasses.

Snippet

```
CS3D
```

DOT

```
DOT [x y]
DOT [x y z]
```

Draws a dot at `[x y]`. This is a very simple drawing command, and isn't really even related to turtle graphics. The dot is drawn using the turtle's pen color, but the turtle itself does not move.
You can also draw a dot in either three dimensional display system. If you are using a 3D display and give a Z coordinate, the coordinate you give will be used. If you are using a 3D

display system and don't give a Z coordinate, DOT uses a Z value of 0, drawing the dot in the computer screen plane.

See "Turtle Positions", earlier in this chapter, for details on the turtle coordinate system.

Note If you give a Z coordinate, and aren't already using one of the 3D display systems, the DOT command will turn on the projection 3D display system.

Snippet

```
DOT [10 10]
```

DOTP

```
DOTP [x y]
```

Outputs TRUE if there is a dot at [x y], and FALSE if there is not.
Technically, DOTP returns TRUE if the pixel is not the same color as the background.
See "Turtle Positions", earlier in this chapter, for details on the turtle coordinate system.

Note DOTP ignores the third dimension. It can tell you if a screen pixel is on or off, but it can't look into three dimensions to see if a point in space is on or off.

This makes a little more sense if you think about how DOTP works. It actually looks at the screen to see if a pixel is on or off, and what color it is. With a 3D display, though, each dot on the screen represents a line running from your eye through the screen pixel, and DOTP has no idea which point on that line is turned on.

Snippet

```
IF DOTP [:x :y] [MAKE "collision "TRUE]
```

FENCE

```
FENCE
```

Confines the turtle to the turtle drawing area, which extends beyond the visible area. For example, in 320 mode, this area extends from -160 to 160 in the X direction, and from 100 to -100 in the Y direction. If the turtle is not in this drawing area when the command is used, it is moved

to [0 0]. Once this command is used, if the turtle tries to move out of the drawing area, an error is flagged and the turtle is not moved.

See WRAP, which wraps the turtle back onto the card when it leaves; and WINDOW, which allows the turtle to roam off of the card.

Snippet

```
FENCE
```

FILL

```
FILL
```

Fills an area with the current pen color. The area can be as small as a single pixel, or as large as the display area, and can be very irregular in shape. The area that is painted is made up of the point at the turtle position and all of the adjacent points that are the same color. The area stops when it hits points of a different color, although it can flow around obstacles.

Note FILL works in two dimensions, but not in three. It always fills an area on the screen.

Snippet

```
TO Triangle :size
REPEAT 3 [FD :size RT 120]
PU
RT 30
FD 10
PD
FILL
BK 10
LT 30
END
```

FORWARD

```
FORWARD distance
FD distance
```

The turtle moves forward distance pixels. If the pen is down, a line is drawn from the old turtle location to the new turtle location.

Snippet

```
REPEAT 5 [FD 30 RT 144]
```

HEADING

```
HEADING
```

Outputs the turtle's heading as a number. The heading is given in degrees, and will always be an integer greater than or equal to 0, and less than 360.

If you are using either 3D display system, HEADING returns a list of three numbers. The first is the standard, 2D heading, which is the longitude angle in the 3D coordinate systems. The second angle is the latitude angle. The third is the roll angle.

See "Turtle Headings", earlier in this chapter, for more information on turtle headings.

Snippet

```
MAKE "oldheading HEADING
```

HIDETURTLE

```
HIDETURTLE
HT
```

Makes the turtle invisible. When the turtle is invisible, the triangle doesn't show up on the screen, but otherwise things work just like they do when the turtle is visible.

Drawing the turtle takes time, so you might want to use this command just before a long series of complicated drawing commands. Depending on what you are drawing, you may see quite a speed improvement.

SHOWTURTLE or SHOWTURTLE3D will make the turtle visible, again.

Snippet

```
HT Draw ST
```

HOME

```
HOME
```

Moves the turtle to [0 0] and sets the turtle's heading to 0. If the pen is down, a line is drawn from the starting position to [0 0].

See "Turtle Positions", earlier in this chapter, for details on the turtle coordinate system.

Snippet

```
HOME
```

LEFT

```
LEFT angle
LT angle
```

Turns the turtle left `angle` degrees.
See "Turtle Headings", earlier in this chapter, for more information on turtle headings.

Snippet

```
REPEAT 6 [FD 30 LT 60]
```

PEN

```
PEN
```

Returns `PENDOWN`, `PENUP`, `PENERASE` or `PENREVERSE`, indicating the state of the turtle pen.

Snippet

```
MAKE "oldpen PEN
```

PENCOLOR

```
PENCOLOR
```

Returns a number from 0 to 7; this is the current pen color.
`SETPC` sets the pen color.
See "Turtle Colors," earlier in this chapter, for a list of the turtle colors.

Snippet

```
SETPC PENCOLOR + 1
```

PENDOWN

PENDOWN
PD

Puts the pen down. When the pen is down, moving the turtle draws a line from the old turtle position to the new turtle position.

See PENUP, PENERASE and PENREVERSE for other pen effects.

Snippet

```
PENDOWN
```

PENERASE

PENERASE
PE

Turns the pen into an eraser. When the pen is erasing, moving the turtle draws a line in the background color, erasing any drawings that are under the line.

See PENDOWN, PENUP and PENREVERSE for other pen effects.

Snippet

```
PENERASE
```

PENREVERSE

PENREVERSE
PX

PENREVERSE is sort of like PENDOWN, since lines are drawn when the turtle is moved, but the line is drawn by reversing pixels. For example, of you draw over an area that is part black and part white, the places that are white will become black, and the places that are black will become white.

PENREVERSE is very handy for animation. If you draw an object with PENREVERSE, then draw it again in the same spot, the object is erased, no matter how complicated the background is.

With a colored pen or colored background, predicting the colors that will show up is pretty tough, but you can still erase any object, even a colored one, by drawing it twice.

See PENDOWN, PENERASE and PENUP for other pen effects.

Snippet

```
Square :40
PX
Square :40
```

PENUP

```
PENUP
PU
```

PENUP lifts the turtle pen. You can still move the turtle, but it won't draw a line.
See PENDOWN, PENERASE and PENREVERSE for other pen effects.

Snippet

```
REPEAT 20 [FD 2 PU FD 2 PD]
```

POS

```
POS
```

Returns the current position of the turtle. The position is returned as a list, with the X coordinate as the first element and the Y coordinate as the second element. For example, the home position is returned as [0 0].

If you are using either 3D coordinate system, POS returns a list if three numbers, instead of two numbers. The third value is the Z coordinate.

See "Turtle Positions", earlier in this chapter, for details on the turtle coordinate system.

Snippet

```
MAKE "old POS
```

RIGHT

```
RIGHT angle
RT angle
```

Turns the turtle right angle degrees.
See "Turtle Headings", earlier in this chapter, for more information on turtle headings.

Snippet

```
REPEAT 4 [FD 30 RT 90]
```

ROLLLEFT

```
ROLLLEFT angle
RLL angle
```

Rolls the turtle left (counterclockwise) `angle` degrees.
See "3D Turtle", earlier in this chapter, for more information on the 3D turtle.

Note If you aren't already using one of the 3D display systems, `ROLLLEFT` will turn on the projection 3D display system.

Snippet

```
REPEAT 4 [Square 40 RLL 90]
```

ROLLRIGHT

```
ROLLRIGHT angle
RLR angle
```

Rolls the turtle right (clockwise) `angle` degrees.
See "3D Turtle", earlier in this chapter, for more information on the 3D turtle.

Note If you aren't already using one of the 3D display systems, `ROLLRIGHT` will turn on the projection 3D display system.

Snippet

```
REPEAT 18 [Draw :H2O RLR 20 ADDFRAME]
```

ROTATEIN

```
ROTATEIN angle
RI angle
```

Turns the turtle into the screen (down) `angle` degrees. Like `LEFT` and `RIGHT`, `ROTATEIN` is based on the current turtle heading, so it you turn the turtle in far enough, it will rotate completely around.

See "3D Turtle", earlier in this chapter, for more information on the 3D turtle.

Note If you aren't already using one of the 3D display systems, `ROTATEIN` will turn on the projection 3D display system.

Snippet

```
REPEAT 4 [FD 30 RI 90]
```

ROTATEOUT

```
ROTATEOUT angle
RO angle
```

Turns the turtle out of the screen (up) `angle` degrees. Like `LEFT` and `RIGHT`, `ROTATEOUT` is based on the current turtle heading, so it you turn the turtle out far enough, it will rotate completely around.

See "3D Turtle", earlier in this chapter, for more information on the 3D turtle.

Note If you aren't already using one of the 3D display systems, `ROTATEOUT` will turn on the projection 3D display system.

Snippet

```
REPEAT 4 [FD 40 RO 90]
```

SCRUNCH

```
SCRUNCH
```

`SCRUNCH` returns the current scrunch factor. See `SETSCRUNCH` for a description of the scrunch factor.

Snippet

```
PR SCRUNCH
```

SETBG

```
SETBG color
```

Sets the background color. CLEAN and CLEARSCREEN fill the card with the background color, and PENERASE causes the turtle to draw lines in the background color. When you start drawing, the background color is white.

See "Turtle Colors," earlier in this chapter, for a list of the turtle colors.

Snippet

```
SETBG 4
```

SETHEADING

```
SETHEADING angle
SETH angle
```

Changes the turtle's heading to angle degrees.

See "Turtle Headings", earlier in this chapter, for more information on turtle headings.

Snippet

```
SETHEADING 45
```

SETHEADING3D

```
SETHEADING3D [longitude latitude roll]
SETH3D [longitude latitude roll]
```

Changes the turtle's heading. The heading works like longitude and latitude on a globe. The first angle, longitude, sets the turtle heading in the screen's view plane. It works the same way as the turtle heading you set with the 2D turtle's SETHEADING command. The second angle, latitude, sets the up-and-down orientation of the turtle. Zero degrees is in the screen plane, just like it is for a 2D turtle. Ninety degrees points straight up, out of the screen. Minus ninety degrees points straight down, into the screen. The last angle rolls the turtle counterclockwise or

clockwise around the heading. Zero degrees is in the screen plane. Ninety degrees rolls the turtle clockwise, so his right legs of the turtle are pointed down. Minus ninety degrees rolls the turtle counterclockwise, so his left legs are pointed straight down.

See "3D Turtle", earlier in this chapter, for more information on the 3D turtle.

Note If you aren't already using one of the 3D display systems, SEATHEADING3D will turn on the projection 3D display system.

Snippet

```
SETHEADING [20 45 45]
```

SETPC

```
SETPC color
```

Sets the pen color. Once you change the pen color, all of the turtle drawing commands will draw lines using the new color. DOT also uses the pen color to decide what color dots should be.

See "Turtle Colors," earlier in this chapter, for a list of the turtle colors.

Snippet

```
SETPC 4
```

SETPENSIZE

```
SETPENSIZE x y
```

SETPENSIZE changes the width and height of lines drawn by the turtle. The default pen size is y = 1, x = 1.

Snippet

```
SETPENSIZE 4 4
```

SETPOS

```
SETPOS [x y]
SETPOS [x y z]
```

Moves the turtle to `[x y]` or `[x y z]`, drawing a line if the pen is down.

If you are using either 3D display system, and only pass a two-element list, `SETPOS` sets the Z coordinate to 0.

See "Turtle Positions", earlier in this chapter, for details on the turtle coordinate system.

Note If you give a z coordinate, and you aren't already using one of the 3D display systems, the `SETPOS` command will turn on the projection 3D display system.

Snippet

```
SETPOS :oldpos
```

SETSCRUNCH

```
SETSCRUNCH scrunch
```

Sets the scrunch factor to `scrunch`.

The scrunch factor is used to adjust the horizontal size of pictures compared to the vertical size. For example, if squares don't look square, you can use the scrunch factor to widen your display or make your display narrower.

When you're using 320 mode graphics, 3D Logo defaults to a scrunch factor of 1. On most monitors, objects will look slightly too narrow – squares will look a little tall for their height, for example. A scrunch factor of about 1.2 will restore things to their proper proportion, although you may have to adjust this number slightly to get perfect pictures. Setting a scrunch factor other than 1 or 2 has a severe disadvantage, though: Drawing is significantly slower.

When you're using 640 mode graphics, the default scrunch factor is 2. For perfect pictures, the scrunch factor should be about 2.4.

Snippet

```
SETSCRUNCH 1.2
```

SETX

SETX x

Moves the turtle horizontally to x, without changing the turtle's Y position. If the pen is down, a line is drawn. The turtle's heading does not change.

See "Turtle Positions", earlier in this chapter, for details on the turtle coordinate system.

Snippet

```
SETX XCOR + 10
```

SETY

SETY y

Moves the turtle vertically to y, without changing the turtle's X position. If the pen is down, a line is drawn. The turtle's heading does not change.

See "Turtle Positions", earlier in this chapter, for details on the turtle coordinate system.

Snippet

```
SETY -YCOR
```

SETZ

SETZ z

Moves the turtle into or out of the screen to z, without changing the turtle's X or Y position. If the pen is down, a line is drawn. The turtle's heading does not change.

See "3D Turtle", earlier in this chapter, for details on the 3D turtle.

Note If you aren't already using one of the 3D display systems, SETZ will turn on the projection 3D display system.

Snippet

```
SETZ 45
```

SHOWNP

```
SHOWNP
```

Returns TRUE if the turtle is visible, and FALSE if it is not visible.
SHOWTURTLE, SHOWTURTLE3D and HIDETURTLE are used to make the turtle visible and invisible.

Snippet

```
IF SHOWNP [HT Draw ST] [Draw]
```

SHOWTURTLE

```
SHOWTURTLE
ST
```

Makes the turtle visible.
HIDETURTLE will make the turtle invisible.

△ **Important** If you are using the stereoscopic 3D display system, SHOWTURTLE turns it off. Be sure to use SHOWTURTLE3D when you are using the stereoscopic 3D display. △

Snippet

```
HT Draw ST
```

SHOWTURTLE3D

```
SHOWTURTLE3D
ST3D
```

Makes the turtle visible, and turns on the 3D stereoscopic display mode. Once this command is used, all turtle drawing commands will create stereo images that appear as true 3D images when you look at the screen with the special 3D glasses.
See "3D Turtle", earlier in this chapter, for details on the 3D turtle.

Note CLEARSCREEN3D turns on the stereoscopic 3D display system. Anything you draw using this display model will look crummy without 3D glasses, but will display as true 3D with the glasses.

Snippet

```
ST3D
```

TOWARDS

```
TOWARDS [x y]
```

Returns a heading that would make the turtle turn towards the point `[x y]`. If you pass this value to `SETHEADING` and draw a line that is long enough, it will pass through `[x y]`.

See "Turtle Positions", earlier in this chapter, for details on the turtle coordinate system.

Snippet

```
MAKE "angle TOWARDS [40 25]
```

WINDOW

```
WINDOW
```

Allows the turtle to roam off of the card. All of the drawing commands will still work, but the results won't be visible.

See `WRAP`, which wraps the turtle back onto the card when it leaves; and `FENCE`, which restricts the turtle to the card.

Snippet

```
WINDOW
```

WRAP

```
WRAP
```

Confines the turtle to a card. If the turtle is off of the card when the command is used, it is moved to `[0 0]`. Once this command is used, if the turtle tries to move off of the screen, it wraps back onto the card. For example, if you set the heading to 90 and draw off of the right edge of the card, the turtle will wrap to the left edge and keep going.

`WRAP` restricts the turtle to the card, not to the display screen. If a card is part-way off of the screen, the turtle can still move to the edge of the card, even if it leaves the display screen.

See FENCE, which restricts the turtle to the card, and WINDOW; which allows the turtle to roam off of the card.

Snippet

```
WRAP
SETHEADING 30
FD 10000
```

XCOR

XCOR

Returns the current X position of the turtle.
See "Turtle Positions", earlier in this chapter, for details on the turtle coordinate system.

Snippet

```
MAKE "x XCOR
```

YCOR

YCOR

Returns the current Y position of the turtle.
See "Turtle Positions", earlier in this chapter, for details on the turtle coordinate system.

Snippet

```
SETPOS [10 YCOR]
```

ZCOR

ZCOR

Returns the current Z position of the turtle. If you are using a 2D turtle, ZCOR always returns zero.
See "Turtle Positions" and "3D Turtle", earlier in this chapter, for details on the turtle coordinate system.

Snippet

```
SETZ -ZCOR
```

Other Drawing Commands

Rectangles

All of the commands you see in this section are based on rectangles, rather than the position of the turtle on the screen. A rectangle is defined by a list, where the list contains the left, top, right and bottom coordinates of the rectangle. The coordinates used match the ones used by the turtle.

For example, to quickly fill a rectangle with black, you could use the commands

```
SETPC 0
PAINTRECT [:left :top :right :bottom]
```

There are two reasons for these commands. Logo's turtle is a great way to draw lines, but it doesn't work well when you want to fill a large area with a color. These commands are very fast compared to filling the same area with the turtle. The second reason to use these commands is that drawing curves and circles in Logo is slow, and the results aren't always very pleasing. The arc and oval drawing commands you see here do the job quickly and well.

The Rectangle-based Drawing Commands

The easiest way to understand these commands isn't to read the descriptions of each command as an individual entity. It's a lot easier to understand these sixteen commands as four different ways to draw four different kinds of shape, for a total of sixteen commands.

The four different ways to draw are:

- erase — Erasing an area works like drawing with the turtle when the pen is in erase mode. All of the pixels in the shape are set to the current background color.
- frame — Framing a shape outlines the shape. It's like tracing the shape using turtle drawing commands. The size of the lines used to outline the shape match the turtle – you can change the size of the lines with the turtle's `SETPENSIZE` command. The turtle's color determines the color of the outline, too.
- invert — Inverting a shape works as if you drew every pixel in the shape with the turtle with a pen color of white and a pen mode of reverse. All black pixels become white, for example, and all white pixels become black. Exactly what happens to the colored

pixels is a little hard to predict without a complicated color map. The overall effect is very useful, though. If you invert the same shape twice, you end up with the same picture you started with. It's a great way to flash and area of the screen!

paint — Painting a shape fills all of the pixels in the shape with the current turtle pen color.

The four different shapes are:

rect — A rect is a rectangle. It's just a box.

oval — An oval is a squashed circle that fits inside of a box defined by a rectangle. Of course, if the rectangle is a square, the oval isn't squashed at all, and you get a circle.

arc — An arc is a slice out of an oval. The slice starts with an angle called `start`, which is measured in degrees, starting at the top of the oval and turning clockwise – just like a turtle heading. A second angle, `angle`, is the size of the slice. It's given in degrees, too.

rrect — This is a rounded rectangle. It's a rectangle with the corners rounded off. The rounded part of the corner is formed by slicing an oval into four chunks, and using one chunk of the oval for each corner. Rounded rectangles come with a `height` and `width` parameter; these are the height and width of the oval that gets cut up to form the corners.

Here's a pair of small procedures that shows what all of these commands actually do on a screen. To see the shapes, type in the procedures and then run `DemoRects`.

```
TO DemoRects
SETBG 15
CS
PU
SETPOS [-50 -50]
PD
SETPC 0
HT
REPEAT 25 [FD 100 RT 90 FD 2 RT 90 FD 100 LT 90 FD 2 LT 90]
MAKE "r [-45 45 -30 30]
ERASEARC :r 45 225
AdjustRect
ERASEOVAL :r
AdjustRect
ERASERECT :r
AdjustRect
ERASERRECT :r 10 10
MAKE "r [-45 20 -30 5]
FRAMEARC :r 45 225
AdjustRect
FRAMEOVAL :r
AdjustRect
```

```
   FRAMERECT :r
   AdjustRect
   FRAMERRECT :r 10 10
   MAKE "r [-45 -5 -30 -20]
   INVERTARC :r 45 225
   AdjustRect
   INVERTOVAL :r
   AdjustRect
   INVERTRECT :r
   AdjustRect
   INVERTRRECT :r 10 10
   MAKE "r [-45 -30 -30 -45]
   PAINTARC :r 45 225
   AdjustRect
   PAINTOVAL :r
   AdjustRect
   PAINTRECT :r
   AdjustRect
   PAINTRRECT :r 10 10
   END

   TO AdjustRect
   MAKE "r (LIST 25 + ITEM 1 :r ITEM 2 :r 25 + ITEM 3 :r ITEM 4
:r)
   END
```

ERASEARC

`ERASEARC rect start angle`

Erases an area on the screen. The area is an arc with the center at the middle of `rect`. The arc starts `start` degrees from the vertical, and is `angle` degrees wide.

See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

ERASEOVAL

`ERASEOVAL rect`

Erases an area on the screen. The area is an oval with edges that touch the edge of the rectangle `rect`.

See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

ERASERECT

```
ERASERECT rect
```

Erases all of the pixels inside of `rect`.
See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

ERASERRECT

```
ERASERRECT rect width height
```

Erases an area on the screen. The area is rectangle formed by `rect`, but the corners of the rectangle are rounded by a quarter-oval. `Height` is the height of the complete oval that is cut up to form the corners, while `width` is the width of the oval.
See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

FRAMEARC

```
FRAMEARC rect start angle
```

Draws the curved outline for an arc. The arc's center is in the middle of `rect`. The arc starts `start` degrees from the vertical, and is `angle` degrees wide. The outline uses the same pen size and color as turtle lines.
See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

FRAMEOVAL

```
FRAMEOVAL rect
```

Outlines the oval formed by `rect`. The outline uses the same pen size and color as turtle lines.
See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

FRAMERECT

```
FRAMERECT rect
```

Draws an outline around the rectangle defined by `rect`. The outline uses the same pen size and color as turtle lines.
See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

FRAMERRECT

```
FRAMERRECT rect width height
```

Draws a rectangle in the area defined by `rect`, but the corners of the rectangle are rounded by a quarter-oval. `Height` is the height of the complete oval that is cut up to form the corners, while `width` is the width of the oval. The outline uses the same pen size and color as turtle lines.

See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

INVERTARC

```
INVERTARC rect start angle
```

Reverses the color of all of the pixels in an area. The area is an arc with the center at the middle of `rect`. The arc starts `start` degrees from the vertical, and is `angle` degrees wide.

See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

INVERTOVAL

```
INVERTOVAL rect
```

Reverses the color of all of the pixels in an area. The area is an oval with edges that touch the edge of the rectangle `rect`.

See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

INVERTRECT

```
INVERTRECT rect
```

Reverses the color of all of the pixels inside `rect`.

See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

INVERTRRECT

```
INVERTRRECT rect width height
```

Reverses the color of all of the pixels in an area. The area is rectangle formed by `rect`, but the corners of the rectangle are rounded by a quarter-oval. `Height` is the height of the complete oval that is cut up to form the corners, while `width` is the width of the oval.

See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

PAINTARC

```
PAINTARC rect start angle
```

Paints all of the pixels in area on the screen, using the current turtle pen color. The area is an arc with the center at the middle of `rect`. The arc starts `start` degrees from the vertical, and is `angle` degrees wide.

See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

PAINTOVAL

```
PAINTOVAL rect
```

Paints all of the pixels in area on the screen, using the current turtle pen color. The area is an oval with edges that touch the edge of the rectangle `rect`.

See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

PAINTRECT

```
PAINTRECT rect
```

Paints all of the pixels in `rect`, using the current turtle pen color.

See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

PAINTRRECT

```
PAINTRRECT rect width height
```

Paints all of the pixels in area on the screen, using the current turtle pen color. The area is rectangle formed by `rect`, but the corners of the rectangle are rounded by a quarter-oval. `Height` is the height of the complete oval that is cut up to form the corners, while `width` is the width of the oval.

See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

Movies

ADDFRAME

ADDFRAME

If the current drawing window is a movie window, ADDFRAME adds a new frame after the frame that is visible and displays the new frame. The position of the turtle does not change, but the new frame is cleared as if CLEAN were used.

If the current drawing window is not a movie window, ADDFRAME does nothing.

Snippet

```
REPEAT 18 [Flag 30 LT 20 ADDFRAME]
```

DELETEFRAME

DELETEFRAME

If the current drawing window is a movie window, and if there are more than one frames in the movie, DELETEFRAME deletes the visible frame and leaves the display on the frame after the one that is visible. If the frame that is deleted is the last frame, the frame before is displayed.

If the current drawing window is not a movie window, or if there is only one frame in the movie, DELETEFRAME does nothing.

Snippet

```
REPEAT 18 [Flag 30 LT 20 ADDFRAME]
DELETEFRAME
```

INSERTFRAME

INSERTFRAME

If the current drawing window is a movie window, INSERTFRAME adds a new frame before the frame that is visible and displays the new frame. The position of the turtle does not change, but the new frame is cleared as if CLEAN were used.

If the current drawing window is not a movie window, INSERTFRAME does nothing.

Snippet

```
REPEAT 18 [INSERTFRAME Flag 30 RT 20]
```

LASTFRAME

LASTFRAME

If the current drawing window is a movie window, and if the visible frame is not the first frame in the movie, LASTFRAME moves to the frame before the visible frame.

If the current drawing window is not a movie window, or if the first frame is being displayed, LASTFRAME does nothing.

Snippet

```
TO TwoFlagMovie
REPEAT 18 [Flag 30 LT 20 ADDFRAME]
REPEAT 18 [LASTFRAME]
PU
SETPOS [50 50]
PD
REPEAT 18 [Flag 10 LT 40 NEXTFRAME]
DELETEFRAME
END
```

NEXTFRAME

NEXTFRAME

If the current drawing window is a movie window, and if the visible frame is not the last frame in the movie, NEXTFRAME moves to the frame after the visible frame.

If the current drawing window is not a movie window, or if the last frame is being displayed, NEXTFRAME does nothing.

Snippet

```
NEXTFRAME
```

PLAY

PLAY

If the current drawing window is a movie window, PLAY plays the movie. The PLAY command works exactly as if the play button on the movie window were pressed.

If the current drawing window is not a movie window, PLAY does nothing.

Snippet

```
TwoFlagMovie
PLAY
```

Fonts

GETFONTINFO

GETFONTINFO

GETFONTINFO returns a list of three integers. In the order they appear in the list, the values represent:

ascent This is the distance from the baseline of the character to the top of the character. For capital letters like K, this is also the height of the letter.

descent This is the distance from the baseline of the character to the bottom of characters that go below the baseline, like "y" and "g".

leading This is the recommended space between two lines.

Use the TEXTWIDTH command to find the width of a line of text.

Note Leading is pronounced like the metal lead, not like "lead a horse." It comes from the not so distant past, when strips of lead were inserted between moveable lead type to increase the space between lines.

Tip Adding ascent and descent gives the height of a line of text. If you add leading, too, you get the distance between two lines of text.

Snippet

```
MAKE "info GETFONTINFO
MAKE "lineSize FIRST :info + ITEM 2 :info + LAST :info
```

SETBACKCOLOR

```
SETBACKCOLOR color
```

SETBACKCOLOR changes the color of the background for text drawn in the turtle window. The colors you can use match the pen color for the turtle in 320 mode, but are limited to black, white, purple and green if you are using 640 mode graphics.

The background for the text includes a rectangle that surrounds the character.

Snippet

```
SETBACKCOLOR 3
```

SETFONTFAMILY

```
SETFONTFAMILY family
```

SETFONTFAMILY changes the font family, which controls the shape of the letters. You can use pretty much any number. If it doesn't match a font in your font folder, Logo picks the closest font it can find.

Here are some of the font numbers you can pick from:

number	name
0	System font (this is the default font)
2	New York
3	Geneva
4	Monaco
5	Venice
6	London
7	Athens
9	Toronto
11	Cairo
20	Times
21	Helvetica
22	Courier
23	Symbol
-2	Shaston

3D Logo

Snippet

```
SETFONTFAMILY 20
```

SETFONTSIZE

```
SETFONTSIZE size
```

SETFONTSIZE changes the size of the letters that are drawn in the turtle window. Font sizes can range from 1 to 255. In general, start with a font size of 12 or 14, then change the size based on what you see.

Tip Some font sizes look better with a particular font than others. That's because the letters are drawn like pictures, but there isn't a separate picture of each letter saved for all 255 possible sizes. The most common predrawn sizes are 9, 10, 12, 14, 16, 24, and 36, although you'll find a number of fonts that don't have all of these sizes, and a few that have more. If you ask for a size that isn't available, Logo does its best to create a font by enlarging or shrinking one of the font sizes that are recorded as pictures. If you don't like the result, try increasing or decreasing the font size.

Snippet

```
SETFONTSIZE 24
```

SETFONTSTYLE

```
SETFONTSTYLE style
```

SETFONTSTYLE changes the way letters are drawn to the screen. There are five different basic styles. Each style has a number; setting the font style to that number picks the style.

number	style
0	plain text
1	**bold**
2	*italic*
4	underline
8	outline
16	shadow

"Plain text" just means you aren't using a special style. All of the other styles can be combined with each other. To combine two or more styles, add all of the style numbers together. For example

```
SETFONTSTYLE 3
```

gives ***bold italic*** text.

Note Some fonts don't support all of these styles. If you can't get the style you want, try switching the font family.

SETFORECOLOR

```
SETFORECOLOR color
```

SETFORECOLOR changes the color of the text drawn in the turtle window. The colors you can use match the pen color for the turtle in 320 mode, but are limited to black, white, purple and green if you are using 640 mode graphics.

Snippet

```
SETFORECOLOR 3
```

TEXTWIDTH

```
TEXTWIDTH word
```

TEXTWIDTH returns the width of a string. GETFONTINFO returns the height.

3D Logo

Snippet

```
TO Center :text :width
LOCAL "x
MAKE "x XCOR
PU
SETX :x + (:width - TEXTWIDTH :text) / 2
PD
TYPE :text
PU
SETX :x
PD
END
```

Desktop Programs

ADDNDA

ADDNDA

Adds all of the available new desk accessories to the first menu.

If you use this command to add desk accessories to the menu bar, you also have to add several special menus. The menus you must add, along with the menu item numbers you must use, are:

id	menu
250	Undo
251	Cut
252	Copy
253	Paste
254	Clear
255	Close

Chapter 6 has an in depth discussion that includes using ADDNDA to support desk accessories.

Snippet

ADDNDA

CHECK

```
CHECK menu item
```

Puts a check mark beside the menu item. `Menu` is the name of the menu that contains the item to check, while `item` is the name of the item.

Snippet

```
CHECK "File "Close
```

CLOSEW

```
CLOSEW name
```

Closes a window. `Name` can be the name of a window or the empty list. If `name` is the empty list, the frontmost window is closed. It is an error to close a window that doesn't exist.

`CLOSEW` returns TRUE if the window was actually closed, and FALSE if the user decided to cancel the close operation.

Snippet

```
CLOSEW "Untitled
```

DESKTOP

```
DESKTOP
```

`DESKTOP` starts a 320 mode desktop program. You get a blank desktop with a blank menu bar. Normally, you'll want to follow this command by creating a menu bar with `MENUBAR`, opening one or more windows with `NEWWINDOW`, and waiting for events with `EVENTS`.

Snippet

```
DESKTOP
```

DESKTOP640

DESKTOP640

DESKTOP640 starts a 640 mode desktop program. It works just like DESKTOP, except that you get a 640 mode display instead of a 320 mode display.

Snippet

```
DESKTOP640
```

DISABLE

DISABLE menu item

Disables a menu item. Menu is the name of the menu that contains the item to disable, while item is the name of the item. Disabled menu items are dimmed, and can't be selected by the user.

Snippet

```
DISABLE "File "Open
```

ENABLE

ENABLE menu item

Enables a menu item. Menu is the name of the menu that contains the item to enable, while item is the name of the item. Enabled menu items are the kind you always use; the text is normal, and the menu item can be selected.

Snippet

```
ENABLE "File "Open
```

EVENTS

EVENTS condition list

EVENTS checks to see if the mouse has been pressed or a key has been pressed. If so, the event is handled. Some events, like those dealing with desk accessories, are handled automatically.

For others, you can set up a Logo command that will get executed when the event is detected. See MENUBAR to find out how to handle a menu event, and NEWWINDOW to see how to handle a click or keypress when one of your windows is the front window.

EVENTS keeps right on handling events as long as condition is true. Generally, you'll test the condition with a single variable, changing the variable when the user picks Quit from the File menu.

List gets executed when your program is idling, waiting for the user to do something. This gives you a chance to do some background processing, like moving objects in a game.

Snippet

```
MAKE "done FALSE
EVENTS NOT :done []
```

GRAPHICS

GRAPHICS

GRAPHICS starts a 320 mode graphics program. You get a completely blank screen that is 200 pixels tall and 320 pixels wide. This screen works exactly like a turtle window.

Snippet

```
GRAPHICS
```

GRAPHICS640

GRAPHICS640

GRAPHICS640 starts a 640 mode graphics program. You get a completely blank screen that is 200 pixels tall and 640 pixels wide. The screen starts with a scrunch factor of 2, so it works like a 320 pixel wide screen.

This screen works exactly like a turtle window.

Snippet

```
GRAPHICS640
```

LOADF

```
LOADF name file
```

Loads a file from disk.

`Name` is the name of the window or the empty list. If `name` is the empty list, the frontmost window is used. It is an error to load to a window that doesn't exist.

`File` is the name of the file to load or the empty list. If you use the empty list, Logo reloads the original file. It is an error to use the empty list for the file name if you have not loaded or saved the file at least once, since Logo won't have a default file name for the file.

`LOADF` returns TRUE if the file was successfully loaded, and FALSE if there was an error trying to read the file.

△ **Important** The window name will change to the name of the file you load. This does not change the name you use to refer to the window. The window name you use as the name parameter to the various Logo commands that take a window name is always the one you used when you opened the window with `NEWWINDOW`. △

Snippet

```
LOADF [] "myfile
```

LOADW

```
LOADW name
```

Loads a file from disk, displaying the file in a window. `Name` can be the name of the window or the empty list. If `name` is the empty list, the frontmost window is used. It is an error to load to a window that doesn't exist.

`LOADW` asks the user for a file name using the same dialog you get when you use the Open command to open a disk file.

`LOADW` returns TRUE if the file was successfully loaded, and FALSE if the user decided not to load a file, or if there was an error trying to read the file.

△ **Important** The window name will change to the name of the file you load. This does not change the name you use to refer to the window. The window name you use as the name parameter to the various Logo commands that take a window name is always the one you used when you opened the window with `NEWWINDOW`. △

Snippet

```
LOADW []
```

MENUBAR

```
MENUBAR list
```

MENUBAR creates a menu bar for a desktop program. `List` is a list of lists, where each of the lists is a different menu on the menu bar.

Each of the menus starts with the name of the menu. In general, it's a good idea to put one space before and after the name for 320 mode programs, and two spaces before and after a menu name for 640 mode programs. Each item in the menu appears as a list, right after the menu name.

Each menu item starts with the name of the menu item. This can be followed by another word, which sets one of several optional menu characteristics. The last thing in a menu item list is a list that Logo will run when the menu is selected. This list can be empty, but if it is, the menu item won't do anything.

There is one special menu name, @. This menu name shows up as the apple menu.

There is also a special menu item name, -. A menu whose name is a single minus sign is drawn as a solid separator line, and can't be selected.

If you use the options field for the menu item, it must be made up of one or more of the following fields, in any order:

option	use
*ab	Defines a keyboard equivalent. You must use two characters after the asterisk. Both will be key equivalents for the menu item. The first will be displayed in the menu when the menu is pulled down. Generally, you should show a capital letter first, followed by its lowercase equivalent.
B	Uses bold text for the menu item.
Cc	The letter after C will be shown in the check mark space. In Logo, you would generally use CHECK and UNCHECK instead of this menu option.
D	Start off with the menu item disabled. In Logo you can use ENABLE and DISABLE to enable and disable the menu item.
I	Uses italics for the menu item.
Nn	This sets up a default menu ID. In Logo, you should only use this option for the special menu items used by desk accessories.
V	Draws a line, like a thin divider, under the menu item.

Here's a sample menu bar, lifted directly from Chapter 6. Chapter 6 gives a complete description of how and why this menu bar was created.

Snippet

```
TO CreateMenus
LOCAL "Bar
LOCAL "Menu
LOCAL "Dash
MAKE "Dash "-
MAKE "Bar []
! [Set up the Apple menu.]
MAKE "Menu [@]
MAKE "Menu LPUT [About... [DoAbout]] :Menu
MAKE "Menu LPUT LIST :Dash [] :Menu
MAKE "Bar LPUT :Menu :Bar
! [Set up the File menu.]
MAKE "Menu [\ File\ ]
MAKE "Menu LPUT [Open... \*Oo [DoOpen]] :Menu
MAKE "Menu LPUT [Close N255\*Ww [DoClose]] :Menu
MAKE "Menu LPUT [Save \*Ss [DoSave]] :Menu
MAKE "Menu LPUT [Save\ As... [DoSaveAs]] :Menu
MAKE "Menu LPUT [Quit \*Qq [MAKE "Done "TRUE]] :Menu
MAKE "Bar LPUT :Menu :Bar
! [Set up the Edit menu.]
MAKE "Menu [\ Edit\ ]
MAKE "Menu LPUT [Undo N250\*Zz []] :Menu
MAKE "Menu LPUT LIST :Dash [] :Menu
MAKE "Menu LPUT [Cut N251\*Xx []] :Menu
MAKE "Menu LPUT [Copy N252\*Cc []] :Menu
MAKE "Menu LPUT [Paste N253\*Vv []] :Menu
MAKE "Menu LPUT [Clear N254 []] :Menu
MAKE "Bar LPUT :Menu :Bar
! [Create the menu bar.]
MENUBAR :Bar
ADDNDA
END
```

NEWWINDOW

```
NEWWINDOW [name kind options rect clickList keyList]
```

NEWWINDOW creates a new window.

Name is the name of the window. This name will be used as the title of the window in the title bar until you load or save a file. After that, the title bar will show the name of the disk file, but you still refer to the window using this original name with all of the Logo commands.

Kind is the kind of the window. Logo supports three different kinds of windows.

kind	use
1	Alert window. Alert windows are a simple box with a heavy outline. They cannot be moved. The About box in the coloring book program from Chapter 6 is an example of an alert window.
2	Turtle window. This is a standard turtle window with scroll bars. It looks and works just like a turtle window in Logo.
3	Turtle window, no scroll bars. This turtle window is a little different from a standard turtle window. It doesn't have scroll bars or a size box, so you can't change its size. The home location for the turtle is in the middle of the window.

`Options` can be 0 or 1. If options is 1, Logo keeps track of whether the contents of the window have changed since the window was opened, or since the last time the window was loaded or saved. If the user closes the window after changes have been made, but before they are safely stored to disk, Logo presents a warning dialog to the user that lets them cancel the close, close the window without saving the changes, or save the changes first. If `options` is 0, `CLOSEW` will close the window without checking for changes.

`Rect` gives the size and location of the window when it is first opened. The rectangle is the inside drawing area of the window, so you need to leave room for the title bar and scroll bars on windows that have them. To position the window, imagine the screen as a large turtle window with the origin smack in the middle. From there, position the rectangle just like you were using `PAINTRECT` or one of the other rectangle based commands.

`ClickList` is a list, which can be empty. If there is anything in the list, the commands will be executed anytime the window is the frontmost window and the user clicks in the drawing area of the window. You can read the mouse position and keep track of the mouse button with `MOUSE` and `BUTTONP`.

KeyList is a list, which can be empty. If there is anything in the list, the commands will be executed anytime the window is the frontmost window and the user uses they keyboard for something that isn't a menu key equivalent. You can read the key with `READCHAR`.

Snippet

```
NEWWINDOW [Untitled 2 1 [5 149 295 18] [DoClick] []]
```

PAGESETUPW

`PAGESETUPW name`

Uses the page setup dialog to get printing options from the user. These options are saved for the current window, and used if the window is printed. `Name` can be the name of a window or the empty list. If `name` is the empty list, the frontmost window is used. It is an error to use a window that doesn't exist.

Snippet

```
PAGESETUPW []
```

PRINTW

`PRINTW name`

Prints a window. `Name` can be the name of a window or the empty list. If `name` is the empty list, the frontmost window is printed. It is an error to print a window that doesn't exist.

Snippet

```
PRINTW []
```

SAVEASW

`SAVEASW name`

Saves a window. `Name` can be the name of a window or the empty list. If `name` is the empty list, the frontmost window is saved. It is an error to save a window that doesn't exist.

Before saving the window, SAVEASW brings up Apple's standard save dialog, so the user can pick a location and name for the file.

△ **Important** The window name will change to the name of the file you save. This does not change the name you use to refer to the window. The window name you use as the name parameter to the various Logo commands that take a window name is always the one you used when you opened the window with `NEWWINDOW`. △

Snippet

```
SAVEASW []
```

SAVEF

`SAVEF name file`

Saves a file to disk.

`Name` is the name of the window or the empty list. If `name` is the empty list, the frontmost window is used. It is an error to use the empty list for the window name if there is no open window.

`File` is the name of the file to save or the empty list. If you use the empty list, Logo saves the current copy of the file to disk. It is an error to use the empty list for the file name if you have not loaded or saved the file at least once, since Logo won't have a default file name for the file.

△ **Important** The window name will change to the name of the file you save. This does not change the name you use to refer to the window. The window name you use as the name parameter to the various Logo commands that take a window name is always the one you used when you opened the window with `NEWWINDOW`. △

Snippet

```
SAVEF [] []
```

SAVEW

`SAVEW name`

Saves the contents of a window. `Name` can be the name of a window or the empty list. If `name` is the empty list, the frontmost window is saved. It is an error to save a window that doesn't exist.

If the window has never been saved, `SAVEW` acts as if you had used `SAVEASW`.

Snippet

```
SAVEW []
```

SELECTW

`SELECTW name`

Selects a window, bringing it to front and making it the active window. `Name` must be the name of a window that is already open. It is an error to select a window that doesn't exist.

Snippet

```
SELECTW "Untitled
```

UNCHECK

```
UNCHECK menu item
```

Removes any check mark from the menu item. If there is no check mark, this command doesn't do anything. `Menu` is the name of the menu that contains the item to uncheck, while `item` is the name of the item.

Snippet

```
UNCHECK "File "Close
```

WINDOWP

```
WINDOWP
```

Returns TRUE if there are any windows open that were created with the NEWWINDOW command, and FALSE if there are no such open windows.

Snippet

```
IF WINDOWP [SAVEASW []]
```

Speech

If you have the speech tools from First Byte installed, Logo can talk. Getting Logo to talk is actually very simple; you just tell it to say a string. There are a lot of options, though. You can control whether Logo uses a male or female voice, how fast Logo talks, and a number of other variables that control the sound of the speech.

Note You must have the First Byte speech tools installed to use these commands. If these tools are not available, these commands don't do anything.

It might seem strange not to get an error or anything, but there is a reason for this. If you want to write a program that uses optional speech, you don't want the program to fail when a friend who doesn't have the speech tools runs the program. If the speech tools flagged an error, the program wouldn't work right unless the speech tools were installed. This way, if

speech isn't available, the program just ignores the speech commands and moves on.

You can add speech to Logo on the Apple IIGS by buying and installing *Talking Tools* from the Byte Works.

DICT

```
DICT word phonetic
```

Logo does a pretty good job of reading English words and converting them into speech. There are a lot of exceptional words in English that aren't pronounced the way they are spelled, though, and Logo needs a little help with these words. `DICT`, combined with `PHONETIC`, gives you an easy way to teach Logo how to say words that aren't pronounced the way they are spelled. Use `PHONETIC` to convert a word that sounds correct into a phonetic string, then give the result, along with the actual spelling of the word, to `DICT`. From that point on, Logo will pronounce the word correctly.

Snippet

```
DICT "Kansas PHONETIC "kansus
```

ERDICT

```
ERDICT
```

`ERDICT` erases all of the words in the dictionary. You put words into the dictionary with `DICT`.

Snippet

```
ERDICT
```

PHONETIC

```
PHONETIC word
```

`PHONETIC` converts `word` to a string of phonetic characters. In most cases, you'll use the result to enter the correct pronunciation for a word into the dictionary using the `DICT` command.

Snippet

```
DICT "Toto PHONETIC towtow
```

PITCH

```
PITCH value
```

PITCH controls the overall frequency of the voice. You can use any number from 0 to 9; if you use a number outside of this range, PITCH adjusts the number to the correct range.

The starting value for PITCH is 5. Higher numbers give a higher pitched voice, which sounds a little squeakier. Low pitch numbers give a lower, rumbling voice.

Snippet

```
PITCH 7
```

SAY

```
SAY word
```

SAY reads word.

▲ **Warning** You should make sure the string isn't too long. Only the first 255 characters or so are actually read. ▲

Snippet

```
SAY "We're\ not\ in\ Kansas\ anymore\,\ Toto.
```

SPEED

```
SPEED value
```

SPEED controls how fast Logo talks. You can use any number from 0 to 9; if you use a number outside of this range, SPEED adjusts the number to the correct range.

The starting value for SPEED is 5. Higher numbers cause Logo to talk faster, while lower numbers cause Logo to talk slower.

Snippet

```
SPEED 7
```

TONE

```
TONE tone
```

TONE adjusts the quality of the voice. You can use two parameters for tone, bass or treble. In both cases, the input is a word, and the case doesn't matter. (BASS works just as well as bass.)

Snippet

```
TONE "bass
```

VOICE

```
VOICE gender
```

VOICE changes the gender of Logo. When Logo starts, it uses a male voice.

```
VOICE "female
```

changes to a female voice, and

```
VOICE "male
```

changes back to the male voice.

The parameter is not case sensitive. MALE works as well as male.

VOLUME

```
VOLUME value
```

VOLUME makes Logo talk louder or quieter. You can use any number from 0 to 9; if you use a number outside of this range, VOLUME adjusts the number to the correct range.

The starting value for VOLUME is 5. Higher numbers cause Logo to talk louder, while lower numbers cause Logo to talk quieter.

Snippet

```
VOLUME 9
```

Workspace Management

BURY

```
BURY word
BURY list
```

BURY buries a procedure or a list of procedures. You can still use a buried procedure, calling it just like you did before it was buried, but the calls that effect all of the procedures in the workspace don't effect buried procedures.

The calls that see unburied procedures, but don't effect procedures once they are buried, are ERALL, ERPS, POALL, POPS, POTS and SAVE.

You can still print or erase a buried procedure, but you have to use one of the commands that requires a specific procedure name as input.

Note BURY only works with your own procedures. You can't bury a primitive.

Snippet

```
BURY "Flag
```

BURYALL

```
BURYALL
```

BURYALL buries all of the procedures, variables and property lists in the workspace.

Snippet

```
BURYALL
```

BURYNAME

```
BURYNAME word
BURYNAME list
```

BURYNAME buries a global variable or a list of global variables. You can still use a buried variable, accessing it with the colon operator or setting the value with MAKE, just like you did before it was buried, but the calls that effect all of the global variables in the workspace don't effect buried variables.

The calls that see unburied variables, but don't effect variables once they are buried, are ERALL, ERNS, POALL, PONS and SAVE.

Snippet

```
BURYNAME [x y z]
```

BURYPLIST

```
BURYPLIST word
BURYPLIST list
```

BURYPLIST buries a property list or a list of property lists. You can still use a buried property list, just like you did before it was buried, but the calls that effect all of the property lists in the workspace don't effect buried property lists.

The calls that see unburied property lists, but don't effect property lists once they are buried, are ERALL, ERPROPS, POALL, PPS and SAVE.

Snippet

```
BURYPLIST "Susan
```

EDIT

```
EDIT word
EDIT list
```

EDIT opens a new edit window. You can give EDIT a single name or a list of names. Each of the names must be the name of an unburied procedure, variable or property list in the workspace. The new window will show all of the objects you listed as a parameter to EDIT.

See "New Edit Window" in Chapter 7 for a technical description of edit windows. Chapter 2 introduces edit windows in a tutorial.

Snippet

```
EDIT "flag
```

EDITS

```
EDITS
```

`EDITS` opens a new edit window. The edit window will contain all of the unburied procedures, variables and property lists currently in the workspace.

See "New Edit Window" in Chapter 7 for a technical description of edit windows. Chapter 2 introduces edit windows in a tutorial.

Snippet

```
EDITS
```

ERALL

```
ERALL
```

Erases all of the global variables, property lists and procedures in the workspace. ERALL does the same thing as these three commands:

```
ERNS
ERPROPS
ERPS
```

Snippet

```
ERALL
```

ERASE

```
ERASE name
ERASE list
```

Erases the named procedures from the workspace. You can use a single name, as in

```
ERASE "Fractal
```

or a list of names, like

```
ERASE [Oval Square Star]
```

ERN

```
ERN name
ERN list
```

Erases the named variables from the workspace. This completely deletes both the variable itself and any value it might have. You can use a single name, as in

```
ERN "Fred
```

or a list of names, like

```
ERN [Fred Sam Psi]
```

ERNS

```
ERNS
```

Erases all of the variables in the workspace.

Snippet

```
ERNS
```

ERPROPS

```
ERPROPS
```

Erases all property lists from the workspace.

Tip There isn't a command to erase an individual property list, but you can get rid of a property list by using REMPROP to remove all of the properties from the property list.

Snippet

```
ERPROPS
```

ERPS

ERPS

Erases all procedures from the workspace.

Snippet

```
ERPS
```

NODES

NODES

Displays the number of free nodes.
This is the number of nodes actually available in the free node pool. To find out how many nodes are actually available, use RECYCLE first to collect all of the free nodes into the free node pool.

Snippet

```
NODES
```

PO

```
PO name
PO list
```

Prints the definitions for all of the named procedures. You can use a single name, as in

```
PO "Fractal
```

or a list of names, like

```
PO [Oval Square Star]
```

POALL

```
POALL
```

Prints all of the global variables, property lists and procedures in the workspace. POALL does the same thing as these three commands:

```
POPS
PPS
PO
```

Snippet

```
POALL
```

PON

```
PON name
PON list
```

Prints the definitions for the named variables. You can use a single name, as in

```
PON "Fred
```

or a list of names, like

```
PON [Fred Sam Psi]
```

PONS

```
PONS list
```

Prints the definitions for all of the global variables. This is a quick way to see what variables are defined, and what the current values are.

Snippet

```
PONS
```

POPS

```
POPS
```

Prints the definitions for all of the procedures in the workspace.

Snippet

```
POPS
```

POT

```
POT name
POT list
```

Prints the header line for all of the named procedures. You can use a single name, as in

```
POT "Fractal
```

or a list of names, like

```
POT [Oval Square Star]
```

The header line is the line that starts with TO and shows the procedure name and parameters.

POTS

```
POTS
```

Prints the header line (the one that starts with TO and shows the procedure name and parameters) for all of the procedures in the workspace.

Snippet

```
POTS
```

PPS

```
PPS
```

Prints all of the property lists in the workspace.

Snippet

```
PPS
```

RECYCLE

```
RECYCLE
```

RECYCLE does garbage collection. Garbage collection scans all of the nodes in the node space, freeing up any nodes that aren't being used.

To understand this command, and why it is useful, you have to understand a little about the way Logo actually works. As you type commands, and as the program runs, Logo is constantly grabbing chunks of memory called nodes. These nodes are used for temporary values, for building lists, for creating strings, and for function return values. If Logo needs a node, but all of them have been used, it does garbage collection. Basically, garbage collection is when Logo takes a moment to clean up after itself, looking at each of the nodes in the node space to see if it's really being used, or was just used and thrown away. The nodes that aren't being used are collected so they can be used again.

The problem with garbage collection is that it takes time, and you don't want it to happen in the middle of a time-critical part of your program. RECYCLE forces Logo to do garbage collection, even if it still has some free nodes. That delays the time until Logo will need to do garbage collection automatically.

Don't overuse RECYCLE! This call takes time, and if you use RECYCLE to often, the whole program will seem slow.

Snippet

```
RECYCLE
```

UNBURY

```
UNBURY word
UNBURY list
```

UNBURY unburies a procedure or a list of procedures.

Note See BURY for more information about buried procedures.

Snippet

```
UNBURY "Flag
```

UNBURYALL

```
UNBURYALL
```

UNBURYALL unburies all of the buried procedures, variables and property lists in the workspace.

Snippet

```
UNBURYALL
```

UNBURYNAME

```
UNBURYNAME word
UNBURYNAME list
```

UNBURYNAME unburies a buried variable or a list of variables.

Note See BURYNAME for more information about buried variables.

Snippet

```
UNBURYNAME [x y z]
```

UNBURYPLIST

```
UNBURYPLIST word
UNBURYPLIST list
```

UNBURYPLIST unburies a buried property list or a list of property lists.

Note See BURYPLIST for more information about buried property lists.

Snippet

```
UNBURYPLIST "Susan
```

Appendix A - Logo Command Summary

Procedures

```
COPYDEF old-name new-name
DEFINE name list
DEFINEDP name
PRIMITIVEP name
TEXT name
TO name parm1 parm2 ...
```

Variables

```
LOCAL name
MAKE name object
NAME name object
NAMEP name
THING name
```

Words and Lists

```
ASCII word
BEFOREP word1 word2
BUTFIRST object
   BF object
BUTLAST object
BL object
   CHAR number
COUNT object
EMPTYP object
EQUALP object1 object2
FIRST object
FLOATP object
FPUT object list
INTEGERP object
ITEM integer object
LAST object
LIST object1 object2
   (LIST object1 object2 object3
    ... )
LISTP object
LOWERCASE word
LPUT object list
MEMBER object1 object2
MEMBERP object1 object2
NUMBERP object
PARSE word
SENTENCE object1 object2
   SE object1 object2
   (SENTENCE object1 object2
     object3 ... )
   (SE object1 object2 object3
     ... )
UPPERCASE word
WORD word1 word2
   (WORD word1 word2 word3 ... )
WORDP object
```

Property Lists

```
GPROP name property
PLIST name
PPROP name property value
REMPROP name property
SETPLIST name list
```

Numbers and Arithmetic

```
ABS value
ARCCOS value
ARCSIN value
ARCTAN value
ARCTAN2 x y
COS angle
DIFFERENCE value1 value2
EXP number
FLOAT number
FORM number width digits
INT number
INTQUOTIENT numerator
  denominator
LN number
POWER x y
PRODUCT number1 number2
   (PRODUCT number1 number2
    number3 ... )
QUOTIENT numerator denominator
RANDOM number
```

```
REMAINDER numerator denominator
RERANDOM
ROUND number
SIN angle
SQRT value
SUM number1 number2
   (SUM number1 number2 number3
    ... )
TAN angle
```

Flow of Control

```
CATCH word list
DOUNTIL list condition
ERROR
GO label
IF condition trueList
   IF condition trueList
falseList
IFFALSE list
   IFF list
IFTRUE list
   IFT list
LABEL label
OUTPUT object
   OP object
REPEAT count list
RUN list
STOP
TEST condition
THROW word
TOPLEVEL
WAIT value
WHILE condition list
```

Logical Operators

```
AND object1 object2
   (AND object1 object2 object3
    ...)
NOT object
OR object1 object2
   (OR object1 object2 object3
    ...)
```

Input and Output

```
BUTTONP
KEYP
MOUSE
PRINT object
   (PRINT object1 object2
    object3 ...)
   PR object
   (PR object1 object2 object3
    ...)
READCHAR
   RC
READCHARS number
   RCS number
READLIST
   RL
READWORD
SHOW object
TEXTIO
TOOT frequency duration
TURTLEIO
TYPE object
   (TYPE object1 object2
    object3)
```

Disk Commands

```
ALLOPEN
CATALOG
CLOSE name
CLOSEALL
CREATEFOLDER name
DIR
ERASEFILE file
FILELEN name
FILEP file
LOAD file
ONLINE
OPEN name
POFILE name
PREFIX
READER
READPOS
RENAME oldname newname
```

```
SAVE file
SETPREFIX prefix
SETREAD name
SETREADPOS position
SETWRITE name
SETWRITEPOS position
WRITEPOS
WRITER
```

Turtle Graphics

```
BACK distance
   BK distance
BACKGROUND
   BG
CLEAN
CLEARSCREEN
   CS
CLEARSCREEN3D
   CS3D
DOT [x y]
   DOT [x y z]
DOTP [x y]
FENCE
FILL
FORWARD distance
   FD distance
HEADING
HIDETURTLE
   HT
HOME
LEFT angle
   LT angle
PEN
PENCOLOR
PENDOWN
   PD
PENERASE
   PE
PENREVERSE
   PX
PENUP
   PU
POS
RIGHT angle
   RT angle
ROLLLEFT angle
   RLL angle
ROLLRIGHT angle
   RLR angle
ROTATEIN angle
   RI angle
ROTATEOUT angle
   RO angle
SCRUNCH
SETBG color
SETHEADING angle
   SETH angle
SETHEADING3D [longitude latitude
 roll]
   SETH3D [longitude latitude
     roll]
SETPC color
SETPENSIZE x y
SETPOS [x y]
   SETPOS [x y z]
SETSCRUNCH scrunch
SETX x
SETY y
SETZ z
SHOWNP
SHOWTURTLE
   ST
SHOWTURTLE3D
   ST3D
TOWARDS [x y]
WINDOW
WRAP
XCOR
YCOR
ZCOR
```

Other Drawing Commands

```
ERASEARC rect start angle
ERASEOVAL rect
ERASERECT rect
ERASERRECT rect width height
FRAMEARC rect start angle
```

```
FRAMEOVAL rect
FRAMERECT rect
FRAMERRECT rect width height
INVERTARC rect start angle
INVERTOVAL rect
INVERTRECT rect
INVERTRRECT rect width height
PAINTARC rect start angle
PAINTOVAL rect
PAINTRECT rect
PAINTRRECT rect width height
```

Movies

```
ADDFRAME
DELETEFRAME
INSERTFRAME
LASTFRAME
NEXTFRAME
PLAY
```

Fonts

```
GETFONTINFO
SETBACKCOLOR color
SETFONTFAMILY family
SETFONTSIZE size
SETFONTSTYLE style
SETFORECOLOR color
TEXTWIDTH word
```

Desktop Programs

```
ADDNDA
CHECK menu item
CLOSEW name
DESKTOP
DESKTOP640
DISABLE menu item
ENABLE menu item
EVENTS condition list
GRAPHICS
GRAPHICS640
LOADF name file
LOADW name
MENUBAR list
NEWWINDOW [name kind options
  rect clickList keyList]
PAGESETUPW name
PRINTW name
SAVEASW name
SAVEF name file
SAVEW name
SELECTW name
UNCHECK menu item
```

Speech

```
DICT word phonetic
ERDICT
PHONETIC word
PITCH value
SAY word
SPEED value
TONE tone
VOICE gender
VOLUME value
```

Workspace Management

```
BURY word
   BURY list
BURYALL
BURYNAME word
   BURYNAME list
BURYPLIST word
   BURYPLIST list
EDIT word
   EDIT list
EDITS
ERALL
ERASE name
   ERASE list
ERN name
   ERN list
ERNS
ERPROPS
ERPS
NODES
```

```
PO name
   PO list
POALL
PON name
   PON list
PONS list
POPS
POT name
   POT list
POTS
PPS
RECYCLE
UNBURY word
   UNBURY list
UNBURYALL
UNBURYNAME word
   UNBURYNAME list
UNBURYPLIST word
   UNBURYPLIST list
```