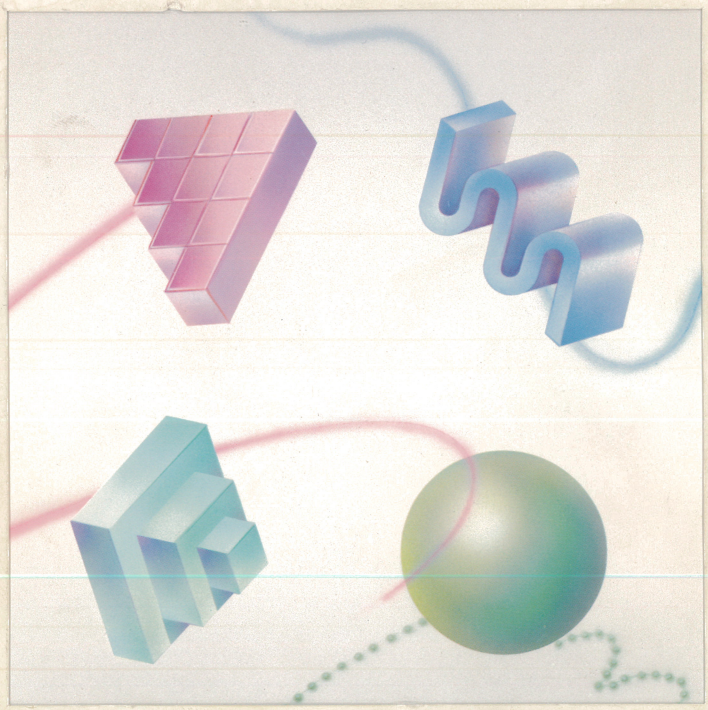
IRIS PROGRAMMING TUTORIAL



SiliconGraphics
Computer Systems

IRIS Programming Tutorial

Version 1.0

SiliconGraphics
Computer Systems

Document Number 007-1103-010

Technical Publications:

Amy B. W. Smith
Steve Locke
Marcia Allen
Robin Florentine
Susan Luttner
Robert Reimann
Diane Wilford

Technical Marketing:

Mason Woo
Michael J. Clark
Thant Tessman
Vince Uttley

Marketing Communications:

Gail Madison

Cover Design:

Lori Blessen

IRIS Programming Tutorial
Version 1.0
Document Number 007-1103-010

About This Tutorial

This tutorial shows you how to create C graphics programs for your IRIS series 2000 or 3000 workstation. Several on-line programs accompany this document to make this a hands-on learning experience. In only one or two hours, you will learn how to write a simple graphics program. In a few days, you will be able to write interactive, 3D graphics programs.

What You Need to Get Started

To begin using this tutorial, you need an IRIS series 2000 or 3000 workstation that is up and running. If your system isn't ready for you to log in, consult your *IRIS Series 3000 Owner's Guide* or *IRIS Workstation Guide, Series 2000.*

To get the most out of this tutorial, you need these skills:

- Some knowledge of the structure of the UNIX operating system. If you don't know anything about UNIX, read *Getting Started with Your IRIS Workstation.*
- A working knowledge of a text editor that runs on your system. To learn the *vi* editor, see *Getting Started with Your IRIS Workstation.*
- Familiarity with the C programming language. Even if you don't know C, you can still get a lot out of this tutorial. Eventually, you'll probably want to consult a C textbook such as *A Book on C* by Al Kelley and Ira Pohl (The Benjamin/Cummings Publishing Company, Inc., 1984).

As you go through the tutorial, keep your *IRIS User's Guide* handy. It contains detailed information about the IRIS Graphics Library.

Contents

1 Learning to Program the IRIS

This is a two-part learning process. You need to understand both the principles of computer graphics, and how the IRIS Graphics Library routines help you apply these principles in your programs.

Using the Graphics Library

This tutorial introduces a simple program that draws a black box. This program is your *template*. You write programs by modifying and enhancing the template as you learn new concepts and routines. By the end of the tutorial, you will have transformed your template into an interactive, 3D graphics program.

The Graphics Library is your toolbox for building and manipulating 3D models. The IRIS thinks in terms of geometry, so you build models by specifying points, lines, polygons, and solids, and you transform models by moving them around in coordinate systems.

The Graphics Library consists of more than 200 routines that vary greatly in sophistication and functionality. One call to the Graphics Library produces a point on the screen; another rotates a model in 3D. The routines fall into four major categories:

- drawing
- color and display
- interaction
- transformation

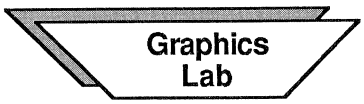
You will use routines in all four categories as you go through this tutorial.

Understanding the Concepts

Because computer graphics is so visual, it's sometimes difficult to learn the concepts from a 2D, non-interactive, textual description (like this book). This is why you are encouraged to use the template described above as you work your way through this tutorial. The text is accompanied by three other types of on-line learning aids:

- Graphics Labs
- Workshops
- Explorations

All of these learning aids help you learn by doing. Be sure to take advantage of them.



Graphics Labs are interactive programs. They help you learn more about a new concept that is introduced in the text. To use a Graphics Lab, run it and follow the on-line instructions. You have access to the source code for the Graphics Labs, but it is very complex. This code will be more useful after you complete this tutorial and move on to more advanced topics. The executable code for Graphics Labs is in the */usr/people/tutorial/c.graphics/online* directory; source code is in */usr/people/tutorial/c.graphics/src/online*.

Workshops are also programs. They help you learn to implement a concept. To use a workshop, read the source code to understand how it works, then compile and run it. The source code for Workshops is in */usr/people/tutorial/c.graphics/src/workshop*; executable code is in */usr/people/tutorial/c.graphics/workshop*.

Explorations

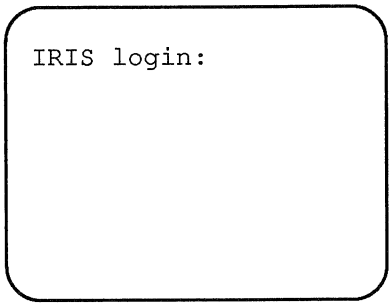
Most Explorations are not programs. They are suggestions for further experimentation with the concepts and routines that you have learned. To use them, make a copy of the template, rename it, and be creative. There are a few programs that are also Explorations. They are a handful of helpful demonstration programs and versions of the template that the text describes. These programs and the template are in */usr/people/tutorial/c.graphics/explore*. Use this directory to work on the template.

2 The IRIS Window Manager

This chapter shows you how to log in and access the files that you use during this tutorial and teaches you how to use the IRIS window manager, *mex*.

Logging In to the Tutor Directory

The starting point for this tutorial is an installed IRIS series 2000 or 3000 workstation, waiting for someone to log in. This is what you should see on your screen:



To begin the tutorial, type:

```
tutor RETURN
```

tutor is a special account set up for the tutorial. When you log in as *tutor*, the current working directory is */usr/people/tutorial/c.graphics*. This directory contains all the sample and demonstration programs you will need for the tutorial.

What Does the Window Manager Do?

When you log in to the *tutor* account, the IRIS starts up its windowing system, called *mex* for Multiple Exposure. *mex* allows you to run several graphics programs at once. The images created by the programs are contained in windows, which you can move around the screen like pieces of paper on a desktop.

Using *mex* has several advantages:

- You can look at the source code for a program in one window while running the program in another window.
- You can run two or more UNIX shells at one time. (This means you can have several windows into the file system at once.)
- You can run two or more versions of a program at the same time in different windows.
- You can keep debugging tools on the screen.

Almost all the programs you write in this tutorial will use the window manager. Although you need to add special *mex* code to your programs, you'll find *mex* makes program development faster and easier.

Note: If you are using an account other than *tutor*, or if you are running the Bourne shell instead of the C shell, you must start up a modified version of *mex* to use this tutorial. To do this, type:

```
whichmex RETURN
```

Using *mex*

Right now the only window you see is the console. It is always on the screen; you cannot delete it, although you can move it, change its shape, or cover it up with other windows.

In the console window, you are logged in to *tutor*. You can execute UNIX commands from here.

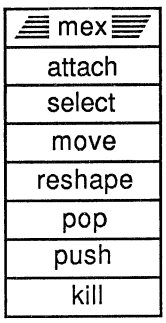
The following sections describe the default user interface for the window manager.

Communicating with *mex*

You communicate with *mex* by pressing the right mouse button. Try out *mex* on the console window.

- Move the mouse so the cursor arrow is located somewhere in the console window.
- Press the right mouse button and hold it down.

You see this menu:



This is the window menu. It appears only when you press the right mouse button while the cursor is located in a window. Each window offers this menu. When you select an option from this menu, it affects only the window under the menu.

- Holding down the right button, move the cursor down the menu.

As you move the cursor, the item it touches is highlighted. **Do not release the button!** If you release the mouse button while an item is highlighted, *mex* carries out the action. This is called "selecting" the item.

- Move the cursor outside the menu. None of the menu items is highlighted. This means *mex* won't do anything when you let up the mouse button.
- While the cursor is outside the menu, release the right button.

The menu disappears and nothing else changes.

Moving a Window

Sometimes you don't want the console to be right in the middle of the screen. To move it, follow these steps:

- Move the cursor so it is in the console window.
- Press and hold the right mouse button.
- Select "move". To do this, move the cursor down the menu until "move" is highlighted, then release the button.

The cursor changes into an image of four outwardly pointing arrows, and a red outline appears on the screen. If you move the cursor around, the red outline moves with it.

- Move the cursor so the red box is where you want to place the console window.
- Press the right mouse button again.

The console is redrawn in the red box. Try moving the console to a couple of different places on the screen. Note that you can move part of it off the screen if you want.

Reshaping a Window

The console takes up most of the real estate on your screen right now, and it's almost time to create another window. You can use the "reshape" option to shrink the console.

- Move the cursor so it is in the console window.
- Hold down the right button and select "reshape".

The cursor now looks like the corner of a window.

- Move the cursor to where you want one of the four corners of the new window to be.
- Hold down the right mouse button to set that corner. Keep the button pressed down and move the mouse diagonally to where you want to place the opposite corner of the console.
- Release the mouse button to set this corner.

The reshaped window is drawn between the two new corners that you set. Try to find out if there are limits to the size of the console window.

The other menu items are important only when there is more than one window on the screen, so you will learn about them in the next section.

Introducing ... Window #2

To see how *mex* really works, you need to open a second window. You can do this in two ways:

- Run a program that creates a window.
- Use the right mouse button and the main *mex* menu to create a window.

You've already seen the menu that appears if you press the right mouse button when the cursor is located in a window. There is a second menu that appears when the cursor is outside of a window — the main *mex* menu.

- Move the cursor outside the console window.
- Press and hold the right mouse button.

You see this menu:

You can always find this menu by pressing the right mouse button outside of a window. You use it to attach input to the background (detach input from all windows), to create new windows, and to stop using *mex*; you do not use it to manipulate existing windows.

- Move the cursor outside the menu and release the mouse button.

Creating a Second Window

Use "new shell" to create another window.

- Move the cursor so it is outside the console.
- Press the right mouse button.
- Select "new shell".

The cursor appears as the corner of a window, just as it did when you reshaped the console.

- Press the right mouse button to set one corner of the new window.
- Move the mouse to where you want the opposite corner to be.
- Release the mouse button to set the corner.

Now you've got a window just like the console. You will be reshaping and creating windows a lot. From now on when you go through the three steps above, think of them as one step: sweeping out a window.

Does the new window overlap the console? If so, you can try the "push" and "pop" menu items. "push" pushes a window behind all other windows; "pop" pops a window on top of the others.

- Move the cursor into the new window.
- Press the right mouse button.
- Select "push".

The console window is redrawn over the new window. Try pushing and popping the two windows. (If they don't overlap, reshape or move one of them so they do.)

Using the Second Window and the Console

When you created a second window, you also started up a new process. You are logged in to *tutor* and you are located in the */usr/people/tutorial/c.graphics* directory. Note that the two windows on the screen right now are windows into the same file system. That is, there is only one *tutor* directory. If you make changes to it in one window, those changes are reflected in the other window.

Now that there are two windows on the screen, you have to specify the one to which you are directing your input. The border of the window to which you are attached is highlighted in red. Now you are attached to the console. Tell *mex* that you want to direct your input (attach) to the new window.

- Move the cursor into the new window.
- Press the right mouse button.
- Select "select".

This pops the new window and directs all input to it. The border of the window becomes red to show it is the active window. If you had selected "attach" instead, the input would have been directed to the new window, but the new window wouldn't have popped to the top.

To list the files in */usr/people/tutorial/c.graphics*, type:

```
ls RETURN
```

Now list the files in the same directory, but use the console window.

- Move the cursor into the console window.
- Press the right mouse button.
- Select "select".

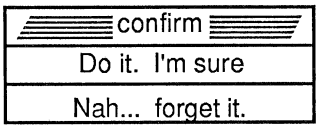
The border of the window becomes red, and it pops to the top. Type the command *ls* in the console window and you see the same listing as in the new window. (If the listings are different, then you're not located in the same directory in both windows. Do a *pwd* to check.)

Deleting a Window

This new window is interesting, but you don't need it now. Go ahead and delete it.

- Move the cursor into the new window.
- Press the the right mouse button.
- Select "kill".

Now you see a new menu, which looks like this:



This new menu protects you if you accidentally select the "kill" menu item. You can cancel the ''kill'' request by selecting ''Nah ... forget it''. This time you really mean it.

- Select "Do it. I'm sure".

The second window disappears.

Running Programs

Now try running some programs. Each program written for the window manager specifies the space it needs on the screen. For example, a program may need a square space, but it can be any size. Or a program may need the whole screen. You will learn about these requests later when you start writing graphics programs.

Running a Graphics Program

If a graphics program requires a specific portion of the screen, the program automatically creates its own window. Otherwise, you need to use the mouse to sweep out an area on the screen for the graphics program.

There is a graphics program called *clock* in the */usr/people/tutorial/c.graphics/explore* directory. To run it, type:

```
clock RETURN
```

Since *clock* doesn't need a specific area of the screen, the only change you see on the screen is in the cursor. It now looks like a corner, as it does when you reshape or create a window.

- Move the cursor to the upper right part of the screen.
- Sweep out a small window.

Now you can do all the things you did with the other windows you had on the screen. Try moving and reshaping the new graphics window. Try hiding it underneath the console window.

Running an Interactive Program

You start up an interactive program the same way you started up *clock*, above. There is one very important difference:

You must always select "attach" or "select" from the program's window menu immediately after you start an interactive program.

When you select "attach" or "select", you direct your input to the window that contains the program. If you forget to do this, you can click the mouse and hit the keyboard all day, but the interactive program will not accept your commands. When you are finished using the program, you must attach to the console so that it can receive your input.

Run the program *cube*.

- Move your cursor into the console window.
- Select "attach" or "select" if the console is not active (if the border is not highlighted).
- Type:

  ```
  cube RETURN
  ```

  A cube appears in the active window.
- Sweep out a window for this program.
- Move your cursor into the new window.
- Select "attach".
- Press the left mouse button and drag the cursor to make the cube move.

Notice that *cube* reads the mouse input only when you are attached to its window.

- Select "kill" to stop the program.

Exiting from *mex*

You now know how to interact with the window manager. You'll find it especially useful when you start writing programs of your own.

If you want to work outside the window manager, you need to exit from *mex*.

- Move the cursor outside all windows.
- Press the right mouse button.
- Select "exit".
- Select "Do it. I'm sure".

Now you've got a textport on the screen where the console used to be. The mouse buttons don't work unless you have a graphics program running. You can type the name of a graphics program into the textport and run it. The textport pops up when the program is finished.

You need to have *mex* running while you use this tutorial. If you are logged in as *tutor*, start *mex* again by typing:

```
mex RETURN
```

If you are using the Bourne shell, or doing the tutorial from an account other than *tutor*, type:

```
whichmex RETURN
```

Summary

- You communicate with the window manager through the right mouse button.
- If the cursor is located over a window, the menu that belongs to that window appears on the screen. This menu shows your options for manipulating that window.
- If the cursor is not over a window, you see the main *mex* menu. You use this menu to open a new window or to exit from *mex*.
- If you choose the ''kill'' or ''exit'' menu items, you must confirm this choice, by selecting ''Do it. I'm sure''.
- When you run an interactive program, you must always remember to attach your input to it. Otherwise, the program does not recognize your input.
- If you exit from *mex* for any reason, you can start it again by typing:

  mex RETURN

- To use this tutorial, Bourne shell users and users logged into an account other than *tutor* must type:

  whichmex RETURN

3 Using the Basic Drawing Routines

This section introduces you to the basic drawing routines. If you're still logged into the *tutor* account from the window manager section — great. Otherwise, go ahead and log in as *tutor*. Make sure you're located in the */usr/people/tutorial/c.graphics* directory.

You are going to learn about the IRIS drawing routines that create these elements:

- points
- lines
- rectangles
- circles
- arcs
- polygons
- text

Drawing images is a two-step process:

1. Select a color.
2. Call drawing routines to draw images on the screen in the color you selected.

Program Template

The C code below carries out this basic algorithm:

- initialize the system (set modes and constants)
- loop until exit
  - draw an image on the screen
  - process input from the user to change the image

You will add to this template to draw the elements listed above. This code is also located in the file */usr/people/tutorial/c.graphics/explore/template.c*.

```
/* This is a template for your graphics programs.
   Note that you start comments with a backslash-
   asterisk and end them with an asterisk-backslash. */

/*-------------------------------------------------------
These first lines attach "include files" to your program.
"gl.h" is a file that contains the IRIS Graphics Library
routines, along with various useful constants. "device.h"
defines a set of common input/output devices.
-------------------------------------------------------*/
#include "gl.h"
#include "device.h"

/*-------------------------------------------------------
"main" is the name of the main function in all C programs.
This function reflects the basic graphics algorithm
shown above.  It calls three other functions,
which are defined below.  "main" loops infinitely; you
exit the program by killing the graphics window with
the right mouse button.
-------------------------------------------------------*/
```

```
main()
{
     initialize();
     drawimage();
     while(TRUE) {
          checkinput();
     }
}

/*-------------------------------------------------------
"initialize" sets various graphics modes (which will be
discussed later).  Right now, all it does is create a
graphics window.  Just before the "winopen" routine,
you can set the characteristics of the window.  In the
beginning, all your programs will use the entire screen.
-------------------------------------------------------*/
initialize()
{
     prefposition(0, 1023, 0, 767);
     winopen("name_of_program");
}

/*-------------------------------------------------------
"drawimage" is the function you will learn about
in this chapter.  It contains the graphics routines
that actually put things on the screen.
The general form of this function is simple:
     select a color and draw something,
     select another color and draw something else,
and so on.  Right now, all this function does is clear the
screen to black.
-------------------------------------------------------*/
drawimage()
{
     color(BLACK);
     clear();
}
```

```
/*-------------------------------------------------------
"checkinput" checks the event queue to see if anything
has happened (for example, has a mouse button been pushed
or has the keyboard been touched?).  REDRAW is a special
event that the window manager puts in the event queue
to tell the program to redraw the graphics window
(e.g., when another window has been moved so that this
one is now visible).  For the moment, don't worry about
how this function works.  You will learn about it in
Chapter 5.
-------------------------------------------------------*/
checkinput()
{
     short val;

     switch(qread(&val)) {
          case REDRAW:
               reshapeviewport();
               break;
          }
}
```

Setting Up Your First Program

Make a copy of the template and call it *draw.c*.

```
cp  template.c  draw.c
```

Then use *vi*, or the text editor of your choice, to eliminate the lengthy comments from *draw.c*.

The only function you need to consider for now is *drawimage*. In the template, *drawimage* does nothing but paint the entire screen black. However, it does include the first two drawing routines: `color` and `clear`.

Selecting a Color

The `color` routine specifies the color to use for drawing. The simplest way to use this routine is to specify one of the eight default colors: BLACK, RED, GREEN, YELLOW, BLUE, MAGENTA, CYAN, or WHITE. These constants are set in the include file *gl.h*. `color` takes one argument, which is either a constant or a color index integer (color indices are explained in Chapter 4).

In the *drawimage* section of the template, `color` uses the color BLACK. Note that you type the name of the color in all capital letters, without quotes or spaces.

Clearing the Window

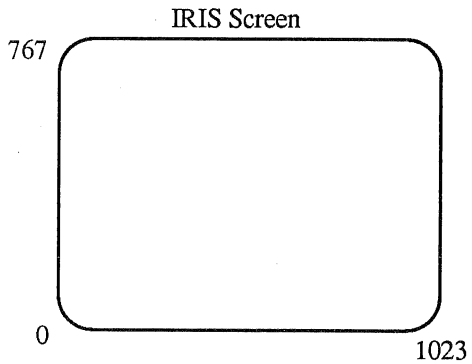
The `clear` routine paints the window in the current color. In *draw.c*, it clears the window to black. `clear` takes no arguments — just an empty set of parentheses.

One common error is drawing something in the same color as its background. For example, if you draw a black box on a black background, it's not going to show up. Always be certain that the color you've selected for drawing is different from the background color.

Drawing Points

After the `clear` routine in *drawimage*, you're going to choose a new color and start drawing some objects. First, try drawing some points. You can think of a point as one pixel on the screen.

Use `pnt2i` to draw a point. It takes two integers as arguments: the *x* and the *y* coordinates of a point. Right now your window takes up the whole screen, so the arguments of `pnt2i` map directly to screen coordinates. Remember that the screen measures 0-1023 along the *x* axis, and 0-767 along the *y* axis.



Edit *drawimage* so it looks like this:

```
drawimage()
{
     color(BLACK);
     clear();
     color(RED);
     pnt2i(50, 50);
}
```

This program draws a little red dot near the lower left corner of the screen. The specified coordinate maps directly to a screen coordinate. This means that the program's window will take up the whole screen and hide the console.

Compile *draw.c*. At the system prompt, type:

```
cc  draw.c  -o  draw  -Zg
```

(See *Getting Started with Your IRIS Workstation* and *IRIS Graphics Programming* for more information on compiling.)

To run the program, type:

```
draw
```

Look carefully for the red dot; it's small. To stop this program, press the right mouse button and select "kill" from the menu.

Exploration

Try drawing in a number of `pnt2i` routines and setting various colors. Make sure the coordinates you choose are located on the screen.

It's tedious to specify the coordinates for each point you want to display. Try generating a number of points:

```
drawimage()
{
int i;
        color(BLACK);
        clear();
        color(GREEN);

for (i=0; i<100; i++)
     pnt2i(10*i, 5*i);
}
```

Note that you need to declare *i* as a variable of type integer at the beginning of the function.

Variations on the Point Routine

Most drawing routines have some variations that you can use to draw different types of pictures. You can vary three characteristics of the routines:

- the number of dimensions in which it draws (two or three);
- the type of numbers it uses (floating point numbers, long integers, or short integers);
- the solidity of the image it draws (outlined or filled).

The `pnt` (point) routine has six variations that you use to specify where drawing should take place. To decide which variation to use, you must make two decisions:

- whether to use 2D or 3D
- whether to use floating point numbers, long integers (24-bit), or short integers (16-bit)

In 3D, you must specify three coordinate components (x, y, and z). In 2D, z defaults to 0 and you only have to specify x and y. For your early programs, use the 2D versions of drawing routines. The 2D routines have `2` appended to the end of their names. For example, the 3D version of the routine for drawing a point with floating point numbers is `pnt`; the 2D version is `pnt2`.

Whether you use floating point numbers or long or short integers depends on the type of data you need for your applications. Use floating point numbers when you want a high degree of accuracy. Use long integers when values will take up more than 16 bits; otherwise, use short integers. The long integer version of the routine ends in the letter `i`, the short version ends in `s`. Use long integers now, since you will be thinking in terms of screen coordinates ($0 \le x \le 1023$ and $0 \le y \le 767$).

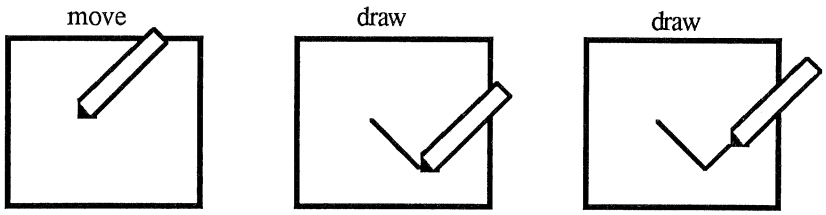
You've already seen the 2D, long integer version of the point drawing routine (`pnt2i`). Here are all six versions:

Routine	Example
`pnt(x, y, z)`	`pnt(20.0, 10.0, 0.0);`
`pnti(x, y, z)`	`pnti(20, 10, 0);`
`pnts(x, y, z)`	`pnts(20, 10, 0);`
`pnt2(x, y)`	`pnt2(20.0, 10.0);`
`pnt2i(x, y)`	`pnt2i(20, 10);`
`pnt2s(x, y)`	`pnt2s(20, 10);`

Drawing Lines

There are two basic routines for drawing lines:

- The **move** routine sets the beginning location for drawing a line. It's like setting down a pen on a piece of paper; no drawing takes place.
- The **draw** routine draws a line from the current drawing position to a new position. It's like drawing a line with the pen. The draw routine also updates the current drawing position to the new position, at the end of the line you've drawn (your pen is positioned at the new location).



Like the **pnt** routine, **move** and **draw** each have six variations:

Routine	Example

`move(x, y, z)`
`movei(x, y, z)`
`moves(x, y, z)`
`move2(x, y)`
`move2i(x, y)`
`move2s(x, y)`
	`move(20.0, 10.0, 0.0);`
	`movei(20, 10, 0);`
	`moves(20, 10, 0);`
	`move2(20.0, 10.0);`
	`move2i(20, 10);`
	`move2s(20, 10);`

`draw(x, y, z)`
`drawi(x, y, z)`
`draws(x, y, z)`
`draw2(x, y)`
`draw2i(x, y)`
`draw2s(x, y)`
	`draw(20.0, 10.0, 0.0);`
	`drawi(20, 10, 0);`
	`draws(20, 10, 0);`
	`draw2(20.0, 10.0);`
	`draw2i(20, 10);`
	`draw2s(20, 10);`

Exploration

Try out **move** and **draw** in *draw.c*. First delete the point routines. Start with a **move** to set the current drawing position. Then you can use **draw** as many times as you want.

To draw a red box, use this *drawimage* function:

```
drawimage()
{
     color(BLACK);
     clear();
     color(RED);

/*-------------------------------------
  Set the pen down at 200, 200.
-------------------------------------*/
     move2i(200, 200);
/*-------------------------------------
  Draw a line from 200, 200 to 200, 300.
-------------------------------------*/
     draw2i(200, 300);
     draw2i(300, 300);
     draw2i(300, 200);
     draw2i(200, 200);
}
```

Changing the Width of Lines

You can vary the thickness of your lines with the **linewidth** routine. The default thickness of a line is one pixel. **linewidth** takes only one argument — the number of pixels wide the lines should be.

You call **linewidth** just before the drawing routines that you want it to affect. To restore the default thickness, call **linewidth** again, and tell it to make the lines one pixel wide. Edit *drawimage* so it looks like this:

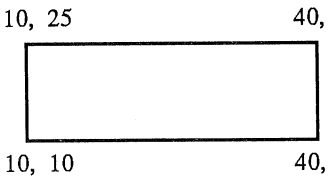
```
drawimage()
{
     color(BLACK);
     clear();
     color(RED);

/*-------------------------------------
Make the lines 3 pixels wide.
-------------------------------------*/
     linewidth(3);
     move2i(200, 200);
     draw2i(200, 300);

     draw2i(300, 300);
/*-------------------------------------
Restore the default width (1).
-------------------------------------*/
     linewidth(1);
     draw2i(300, 200);
     draw2i(200, 200);
}
```

Drawing Rectangles

Using **rect**, you can draw two types of rectangles:

- outlined rectangles (four lines)
- filled rectangles (solid blocks)

A rectangle is defined by any two opposite corners.



You can specify this rectangle either way:

```
recti(10, 10, 40, 25);
recti(10, 25, 40, 10);
```

Because rectangles are two-dimensional shapes, the IRIS assumes that the *z* value is zero (0). There are no 3D versions of **rect**.

Routine	Example

`rect(x1, y1, x2, y2)`
`recti(x1, y1, x2, y2)`
`rects(x1, y1, x2, y2)`
	`rect(20.0, 10.0, 300.0, 200.0);`
	`recti(20, 10, 300, 200);`
	`rects(20, 10, 300, 200);`

`rectf(x1, y1, x2, y2)`
`rectfi(x1, y1, x2, y2)`
`rectfs(x1, y1, x2, y2)`
	`rectf(20.0, 10.0, 300.0, 200.0);`
	`rectfi(20, 10, 300, 200);`
	`rectfs(20, 10, 300, 200);`

Here's *drawimage* with some rectangle routines:

```
drawimage()
{
     color(BLACK);
     clear();

     color(BLUE);
     recti(300, 300, 500, 500);
     rectfi(500, 500, 600, 800);
}
```

Drawing Circles

Drawing circles is similar to drawing rectangles. The IRIS assumes that $z = 0$, and circles can be outlined or filled. You specify the center point and a radius. You can use floating point numbers, long integers, or short integers for the arguments.

Routine	Example

`circ(x, y, radius)`
`circi(x, y, radius)`
`circs(x, y, radius)`
	`circ(20.0, 20.0, 100.0);`
	`circi(20, 20, 100);`
	`circs(20, 20, 100);`

`circf(x, y, radius)`
`circfi(x, y, radius)`
`circfs(x, y, radius)`
	`circf(20.0, 20.0, 100.0);`
	`circfi(20, 20, 100);`
	`circfs(20, 20, 100);`

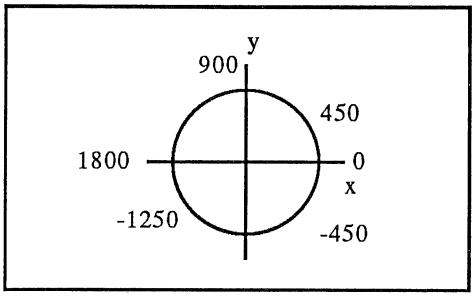
Try this in *drawimage*:

```
drawimage()
{
     color(BLACK);
     clear();

     color(RED);
     circi(900, 600, 40);
     circfi(800, 600, 40);
}
```

Drawing Arcs

Arcs are like circles: two-dimensional, filled and unfilled, specified by a center point and a radius. However, you also need to specify how much of an arc you want to draw. To do this, you use a starting angle and an ending angle.

The angles for an arc are measured in *tenths* of a degree, and they are always integers. Angles are measured from the *x*-axis in a counterclockwise direction. The three o'clock position equals zero. Twelve o'clock is 90 degrees or 900 tenths of a degree.



To draw two arcs, try this sample function:

```
drawimage()
{
     color(BLACK);
     clear();

     color(BLUE);
     arci(200, 100, 100, 0, 500);
     arcfi(500, 100, 100, 900, 1200);
}
```

Here are the arc routines:

Routine	Example

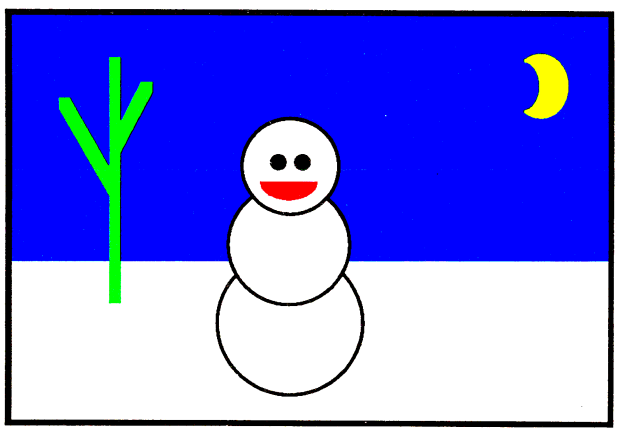
`arc(x, y, radius, startang, endang)`
`arci(x, y, radius, startang, endang)`
`arcs(x, y, radius, startang, endang)`
	`arc(10.0, 10.0, 40.0, 0, 900);`
	`arci(10, 10, 40, 0, 900);`
	`arcs(10, 10, 40, 0, 900);`

`arcf(x, y, radius, startang, endang)`
`arcfi(x, y, radius, startang, endang)`
`arcfs(x, y, radius, startang, endang)`
	`arcf(10.0, 10.0, 40.0, 0, 900);`
	`arcfi(10, 10, 40, 0, 900);`
	`arcfs(10, 10, 40, 0, 900);`

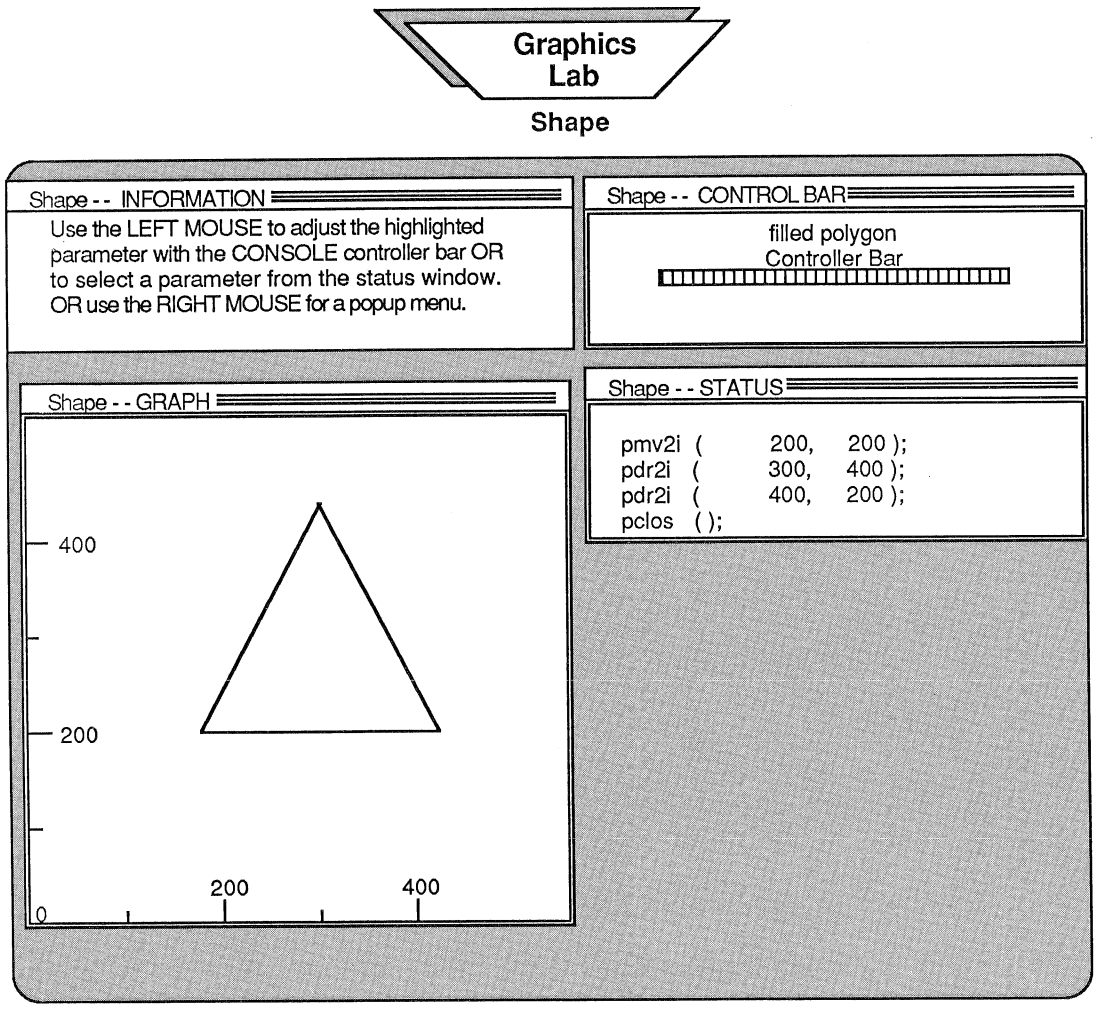
Exploration

Experiment with different arc routines and different angles. Try negative angles.

Make a copy of *draw.c* and call it *snow.c*. Tie together all the routines you have learned so far to draw this winter scene:



You will have a chance to add to *snow.c* later, so you may want to save it.



Change directories (*cd*) to */usr/people/tutorial/c.graphics/online*. To run the Graphics Lab called *shape*, type:

```
shape
```

You choose several different shapes for it to draw. As it draws a shape it also displays the routine it is executing. Compare the routines with what appears on the screen. (Your IRIS must have at least 12 bitplanes to run this Lab.)

Drawing Polygons

As with rectangles, circles, and arcs, there are two types of polygons: filled and outlined.

Outlined Polygons

A polygon is a closed plane figure with three or more straight sides. You already know one way to draw an outlined polygon — using **move** and **draw**. Move to the first vertex of the polygon. Draw a line to the next vertex. Keep drawing lines until you return to the original vertex.

The **poly** routine also draws an outlined polygon. It takes two arguments: the number of vertices in the polygon and the name of the array that specifies those vertices. **poly** comes in all the normal variations: 2D and 3D; floating point number, long integer, and short integer. Here is a sample routine:

```
poly2i(4, parray);
```

This poses a new problem: how do you set up an array of coordinates?

Setting Up Arrays of Coordinates

First of all, since *parray* is a variable, you need to declare it at the beginning of your function. *parray* contains coordinates which are one of three types: floating point, long integer, or short integer. The Graphics Library is set up with special type definitions for these coordinates: **Coord**, **Icoord**, and **Scoord**, which are abbreviations for floating point, long integer and short integer coordinates. You're using long integers, so declare *parray* as **Icoord**.

You also need to describe the size of the array. It's going to contain four points, each of which has two components (x and y). This means it's a 4x2 array. You put this information in the declaration line. Note that you need only specify all the vertices — the IRIS automatically closes the polygon by connecting the first and last coordinates.

So, before any of the drawing routines, at the beginning of your function, declare the array:

```
Icoord parray[4][2];
```

If you want to make a 3D array, you would instead declare **`Icoord parray[4][3];`**. This tells the IRIS that there are four vertices, each made up of three components. If the 3D coordinates were floating point numbers, you would declare **`Coord parray[4][3];`**.

Now you need to load values into the array. You can do this in two ways:

- You can set each element of the array individually. Note that array indices start with 0, not 1.

```
/*-----------------------------------
  The first coordinate is (600, 100).
-----------------------------------*/
parray[0][0]=600;
parray[0][1]=100;

/*-----------------------------------
  The second coordinate is (700, 200).
-----------------------------------*/
parray[1][0]=700;
parray[1][1]=200;

/*-----------------------------------
  The third coordinate is (800, 200).
-----------------------------------*/
parray[2][0]=800;
parray[2][1]=200;

/*-----------------------------------
  The fourth coordinate is (650, 100).
-----------------------------------*/
parray[3][0]=650;
parray[3][1]=100;
```

- Alternatively, you can set all the elements in the declaration of the array. When you use this method, you must declare the array as *static*. This means the local variables will still contain the values assigned to them in this procedure the next time the procedure is called.

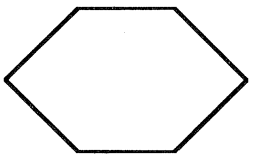
```
static Icoord parray[4][2] = {
                            {600, 100},
                            {700, 200},
                            {800, 100},
                            {650, 100}
                                    };
```

This *drawimage* function uses the first type of array declaration to draw a red outlined polygon.

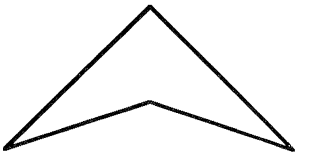
```
drawimage()
{
     Icoord parray[4][2];

     parray[0][0]=600;
     parray[0][1]=100;
     parray[1][0]=700;
     parray[1][1]=200;
     parray[2][0]=800;
     parray[2][1]=200;
     parray[3][0]=650;
     parray[3][1]=100;

     color(BLACK);
     clear();

     color(RED);
     poly2i(4, parray);
}
```

Filled Polygons

If you substitute `polf2i` for `poly2i` in *drawimage*, you'll get a filled polygon instead of an outlined one. All other steps remain the same. The one thing to remember is to draw only convex filled polygons. In a convex polygon, when you connect any two vertices with a line, the line falls entirely within the polygon. If you try to draw a concave filled polygon, the system may become confused about what is the inside and what is the outside of the polygon. It might try to fill the entire screen instead of the interior of the polygon.



convex polygon



concave polygon

Another way to draw filled polygons is to use the `pmv` (polygon move) and `pdr` (polygon draw) routines. `pmv` specifies the first vertex of a filled polygon. `pdr` routines specify the remaining vertices (just like using `move` and `draw` for lines). To close the polygon after specifying all of its vertices, use the `pclos` (polygon close) routine. Here is an example:

```
drawimage()
{
     color(BLACK);
     clear();

     color(RED);

     pmv2i(700, 400);
     pdr2i(700, 500);
     pdr2i(800, 500);
     pdr2i(900, 300);
     pclos();
}
```

Here are all the polygon routines:

Routine	Example

`poly(n, parray)`
`polyi(n, parray)`
`polys(n, parray)`
`poly2(n, parray)`
`poly2i(n, parray)`
`poly2s(n, parray)`
	`poly(3, parray);`
	`polyi(3, parray);`
	`polys(3, parray);`
	`poly2(3, parray);`
	`poly2i(3, parray);`
	`poly2s(3, parray);`

`polf(n, parray)`
`polfi(n, parray)`
`polfs(n, parray)`
`polf2(n, parray)`
`polf2i(n, parray)`
`polf2s(n, parray)`
	`polf(3, parray);`
	`polfi(3, parray);`
	`polfs(3, parray);`
	`polf2(3, parray);`
	`polf2i(3, parray);`
	`polf2s(3, parray);`

`pmv(x, y, z)`
`pmvi(x, y, z)`
`pmvs(x, y, z)`
`pmv2(x, y)`
`pmv2i(x, y)`
`pmv2s(x, y)`
	`pmv(20.0, 10.0, 0.0);`
	`pmvi(20, 10, 0);`
	`pmvs(20, 10, 0);`
	`pmv2(20.0, 10.0);`
	`pmv2i(20, 10);`
	`pmv2s(20, 10);`

`pdr(x, y, z)`
`pdri(x, y, z)`
`pdrs(x, y, z)`
`pdr2(x, y)`
`pdr2i(x, y)`
`pdr2s(x, y)`
	`pdr(20.0, 10.0, 0.0);`
	`pdri(20, 10, 0);`
	`pdrs(20, 10, 0);`
	`pdr2(20.0, 10.0);`
	`pdr2i(20, 10);`
	`pdr2s(20, 10);`

`pclos()`	`pclos();`

Exploration

If you saved *snow.c*, put a hat on the snowman, and add some evergreens to the background. Note that both of these shapes look like concave, filled polygons. You can construct the shapes from convex filled polygons by stacking triangles and rectangles.

Workshop

Look over the code for *diamond1.c* in the */usr/people/tutorial/c.graphics/src/workshop* directory. It draws a baseball diamond. For now it's a 2D, non-interactive program. In each workshop, you will add to this basic program, until it contains all of the concepts and routines that you learn in this tutorial. The end result will be a 3D baseball diamond with simulated movement and perspective.

Text

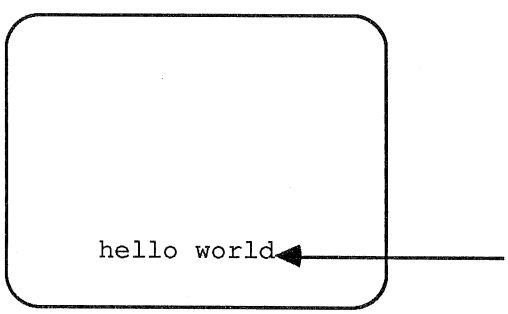
The IRIS treats text in a graphics port exactly like a picture, so you draw text on the screen like any other element.

cmov sets the character location. It comes in the usual variations:

Routine	Example
`cmov(x, y, z)`	`cmov(20.0, 10.0, 0.0);`
`cmovi(x, y, z)`	`cmovi(20, 10, 0);`
`cmovs(x, y, z)`	`cmovs(20, 10, 0);`
`cmov2(x, y)`	`cmov2(20.0, 10.0);`
`cmov2i(x, y)`	`cmov2i(20, 10);`
`cmov2s(x, y)`	`cmov2s(20, 10);`

Use **cmov2i** so that your coordinates match up with the screen coordinates.

charstr draws text on the screen. It takes a character string as its argument. After **charstr** draws the text, the new character location is at the end of the string.




```
cmov2i(300, 200);
charstr("hello world");
```

If you call another `charstr` without a `cmov`, the string will start here.

To draw text, follow these three steps:

1. Set the color of the text with **color**. Make sure it's different from the background color. (Make the background blue for this example).
2. Set the character position with **cmov2i**.
3. Draw the text string with **charstr**.

Remove all of the drawing routines from *drawimage* and add these text routines.

```
drawimage()
{
     color(BLUE);
     clear();

/*-------------------------------------------------
Set the drawing color to red.
-------------------------------------------------*/
     color(RED);
/*-------------------------------------------------
Move the character position to (300,380).
-------------------------------------------------*/
     cmov2i(300, 380);

/*-------------------------------------------------
Draw the first string.
-------------------------------------------------*/
     charstr("The first line is drawn ");

/*-------------------------------------------------
Draw the second string (which starts where the
first ended).
-------------------------------------------------*/
     charstr("in two parts. ");
```

```
/*-------------------------------------------------
Move the character position down 12 pixels and
back to the beginning of the original line.
-------------------------------------------------*/
     cmov2i(300, 368);

/*-------------------------------------------------
Draw the last string.
-------------------------------------------------*/
     charstr("This line is 12 pixels lower. ");
}
```

Exploration

Add a tacky billboard to your winter scene in *snow.c*.

Summary

- Make sure you know what the current drawing color is, and that it's different from the background color.
- Make sure you give the different variations of the routines the appropriate arguments.
- Refer to *IRIS User's Guide* for lists of all the variations for the Graphics Library routines.
- Always specify angles in tenths of degrees, so 30 degrees = 300 tenths.
- Don't use a single array to draw concave, filled polygons. You can usually construct them from convex, filled polygons.

4 Using Color and Display Modes

Each time you drew an object in the last chapter, you changed the values that the IRIS had stored in its image memory. These changes show up on the screen when the IRIS uses the values to illuminate screen pixels. This chapter discusses how you store values in the image memory, and how the IRIS uses those values.

Understanding RGB Intensities

The images that appear on the screen consist of thousands of pixels. Each pixel on the screen is a cluster of three tiny phosphors. When the system shoots beams of electrons at the phosphors, they glow for a moment. One phosphor glows red; one glows green; one glows blue.

If the beams don't shoot any of the three phosphors, the pixel looks black; it doesn't glow at all. If they shoot only the blue phosphor, you see a bright blue dot. If they shoot the red phosphor a little and the blue phosphor a little, you see a purple dot. And if they shoot all three phosphors full blast, you see a white dot.

Each of the three components, red(R), green(G), and blue(B), has 256 possible intensities (0 to 255). 0 tells the hardware not to shoot the particular phosphor; 255 tells it to shoot the phosphor with full intensity. You need 8 bits (bitplanes) to represent all 256 (2^8) intensities. Since you have three phosphors (R, G, and B), and 8 bits per phosphor, you need 24 bits (3 phosphors x 8 bits) to represent all possible colors for one pixel. This also means you have 2^{24} or roughly 16.8 million colors to choose from.

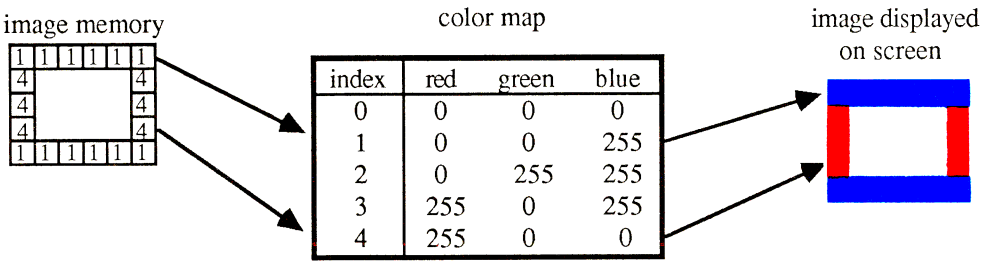
Exploration

Run the program *showmap* which is in the */usr/people/tutorial/c.graphics/explore* directory. It shows you all the colors that the color map contains.

It seems that your image memory needs 24 bitplanes (8 bits for each R, G, and B) to provide colors for a given pixel. But what if you have only 12 bitplanes of image memory? Or 8? How can you use the full range of colors? As it turns out, you don't need to store 24-bit RGB values in your image memory. Most people don't do this, even if they have the full complement of bitplanes; they use indices into a color map.

Color Maps

The IRIS color map is another way to handle color. This map is a table of 24-bit RGB values. Every 24-bit value has its own index number. You store the index number in the bitplanes and the IRIS looks it up in the color map to find out which color to display. If you have 12 bitplanes, you can put 4096 colors in the color map.



The Default Color Map

This table shows the default color map. Instead of using a number as an argument, you can use a constant (like BLACK or RED).

Index	Name	Red	Green	Blue
0	BLACK	0	0	0
1	RED	255	0	0
2	GREEN	0	255	0
3	YELLOW	255	255	0
4	BLUE	0	0	255
5	MAGENTA	255	0	255
6	CYAN	0	255	255
7	WHITE	255	255	255
others	unnamed	undefined		

Loading the Color Map

You define any of the color map entries (including the original eight) with the **mapcolor** routine. **mapcolor** takes four arguments: a color map index (an integer between 0 and 4095), one integer between 0 and 255 for the red value, one integer for green, and one integer for blue.

Try some **mapcolor** routines. Make a copy of the template and call it *color.c*. Now follow two steps:

1. Set index 8 of the color map to 150 units of red, 0 units of green, and 200 units of blue. Edit the *initialize* function. After **winopen** in *initialize*, add this line:

```
mapcolor(8, 150, 0, 200);
```

2. Delete all drawing routines from *drawimage*, then change the current color from BLUE (which is color map index 4) to 8. Now the screen will be cleared to color 8 when you run the program.

initialize and *drawimage* should now look like this code:

```
initialize()
{
     prefposition(0, 1023, 0, 767);
     winopen("color.c");

     mapcolor(8, 150, 0, 200);
}
drawimage()
{
     color(8);
     clear();
}
```

Compile and run *color.c*. Before you kill the program, push the *color.c* window so that you can see the *showmap* window you created before. If you look in the bottom row, you can see that entry 8 is now defined.

All the programs you run share the same color map. So, if you change any entry in the map, programs that use that entry will display a different color. This is important if you're writing a number of programs that run at the same time.

Creating a Color Ramp

Most of the time you don't need a lot of different colors. It's more likely that you'll need a few different colors, plus a range of shades. For example, you can create a color ramp that contains several shades of red.

Use the `mapcolor` routine to load part of the color map with shades of red. Leave the eight default colors for now. You can load up to 256 shades of pure red. In the example below you'll load 240 shades of red into color map indices 8 through 247.

To make the algorithm as simple as possible, make the red RGB component equal to the color map index. That is, entry 8 is 8-0-0 , entry 9 is 9-0-0, entry 10 is 10-0-0, and so on. Entry 247 is 247-0-0. All the colors will contain only red components. 8-0-0 looks black on the screen, so you don't need to enter any red components less than 8. 247-0-0 isn't quite the brightest red possible (which is 255-0-0), but it's pretty close.

You need to change the *initialize* function in *color.c*. (These changes are shown in the code on the next page.)

1. Declare *i* as an integer variable. This lets you create the map with an iterative loop, rather than 240 separate `mapcolor` routines.
2. Delete the old `mapcolor` routine and create the new statement listed below. If you are not familiar with the for statement, refer to a C programming book (such as *A Book on C*).
3. Substitute another color index, like 125, for index 8 in the `color` routine in *drawimage*.

Remember, all this program does is clear the screen to a specified color. To create a color ramp of reds, edit *initialize* and *drawimage* in *color.c* so they look like the code below.

```
initialize()
{
/*-----------------------------------------------------
Declare i as an integer variable.
-----------------------------------------------------*/
     int i;

     prefposition(0, 1023, 0, 767);
     winopen("color.c");

/*-----------------------------------------------------
Load color map locations 8-247 with RGB values
of increasing red intensity.
-----------------------------------------------------*/
     for (i=8; i<248; i++)
          mapcolor(i, i, 0, 0);
}

drawimage()
{
/*-----------------------------------------------------
Set the color to color map index 125.
-----------------------------------------------------*/
     color(125);
     clear();
}
```

Compile and run *color.c*. Your screen turns dark red. Use the right mouse button to bring up the window menu, and push the *color.c* window behind the console and the *showmap* window. *showmap* displays your new colors. Remember, all the programs you run share the same color map. So, when you change the color map in *color.c*, *showmap* reflects those changes.

Move the cursor over the red part of the screen again and kill *color.c*.

Exploration

Now you can use some routines from the previous chapter to display your color ramp. Make a copy of *color.c* and call it *sunset.c*.

Try to draw a sunset with the shades of red. Use a for loop to increase gradually the size of your arcs and decrease the brightness of the color.

Also, look over the program *ramp.c* in */usr/people/tutorial/c.graphics/explore*. It creates the red color ramp and displays each different shade in a 20x20 pixel box that is labeled with each shade's color map index. Run it and take a look at the result.

Workshop

Look over *colored2.c* in the */usr/people/tutorial/c.graphics/src/workshop* directory. It adds some life to the baseball diamond.

Display Modes

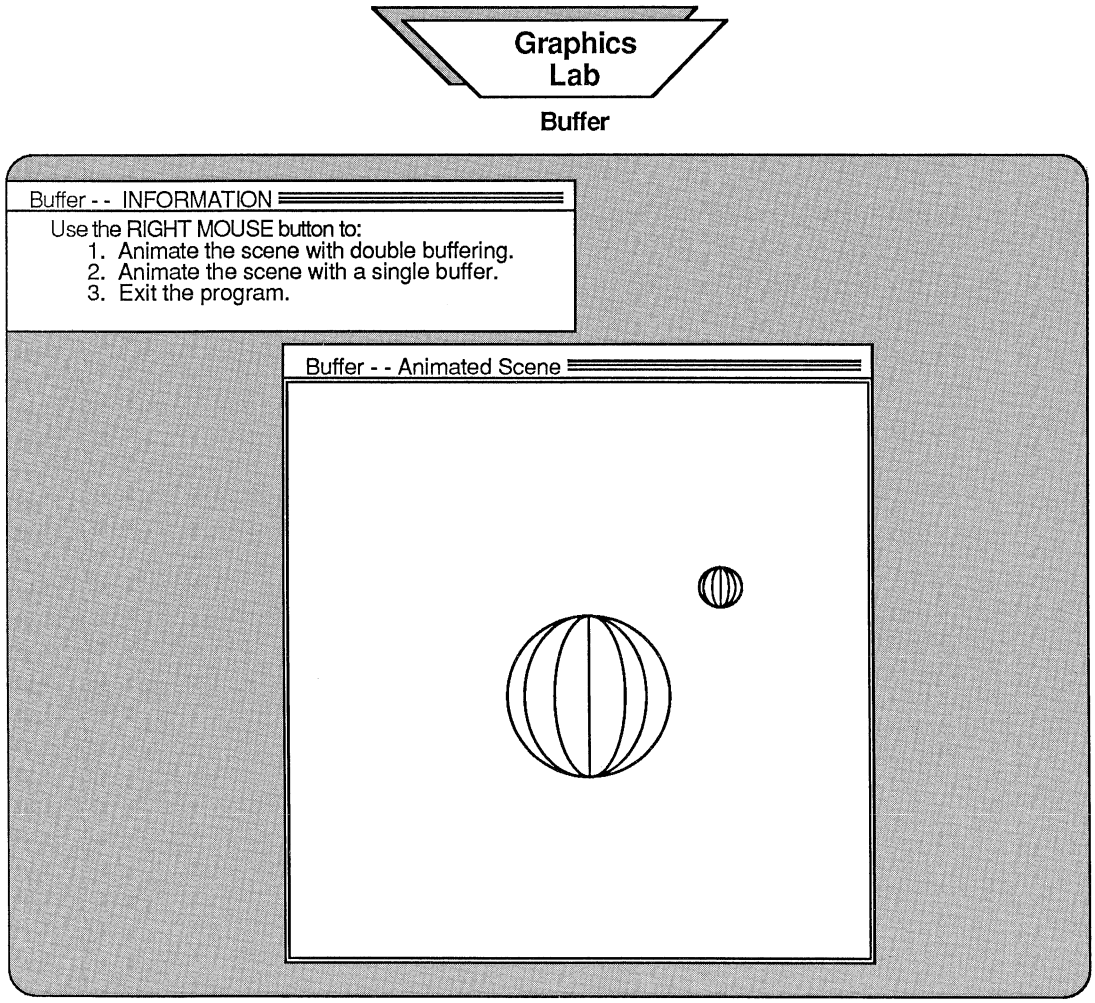
The IRIS uses its image memory in two ways: to store color and image information, and to display color images. This means the IRIS often writes new values into the bitplanes at the same time that it uses those values to display images. It can't actually do both things at once — the IRIS first stores the new values, then displays them. This slight time lag sometimes results in flickering or half-drawn images.

This is a problem mainly in programs that simulate motion, because they involve a lot of rapid redrawing and updating of images. Fortunately, the IRIS offers a good solution.

Double Buffer Mode

Here's the basic concept:

- Divide your image memory (your bitplanes) in half.
- Tell the hardware to display whatever image is stored in one half, while you write values for a new image in the other half.
- When the new image is complete, tell the IRIS to swap halves, or *buffers* — that is, tell it to display the new image and start updating the information contained in the other half.
- Create and swap again, and again, and again.



Run *buffer* to see how much better things look in double buffer mode — no flashing. (Your IRIS must have at least 12 bitplanes to run this Lab.)

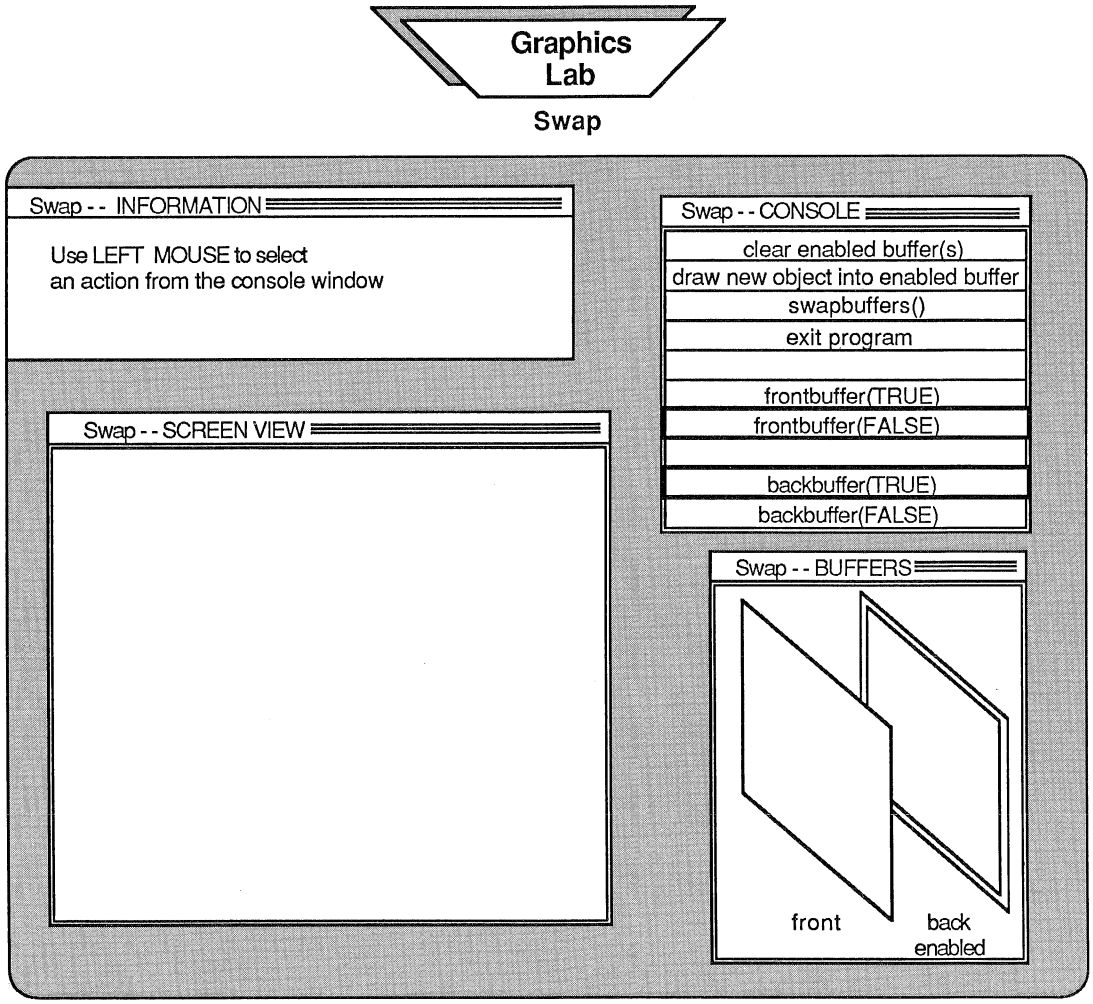
Double buffer mode improves the quality of your images, but it also imposes some limitations. If you have only eight bitplanes, each of your buffers uses only four bitplanes. This means that in double buffer mode the color map can store only 16 colors.

Using Double Buffering

When you use double buffering, you suddenly have two buffers — the *front buffer* and the *back buffer*. The IRIS lets you draw into either, neither, or both of these buffers.

`frontbuffer(TRUE)` means the front buffer stores color indices when you call drawing routines. `frontbuffer(FALSE)` means the front buffer doesn't store anything. `backbuffer(TRUE)` and `backbuffer(FALSE)` work the same way. The default is `frontbuffer(FALSE)` and `backbuffer(TRUE)`. The front buffer is the one that the system displays on the screen, so you usually write to (draw in) the backbuffer.

When the new image is completely drawn, you swap buffers with `swapbuffers` (no arguments). The new image in the back buffer is put into the front buffer and the old image in the front buffer is put in the back buffer. (Actually, no values are moved; the labels of the buffers are switched.) You continue to store new values in the back buffer until you are ready to swap again.



Run *swap*. It shows you which images are displayed and where values are written depending on which buffer is enabled. (Your IRIS must have at least 12 bitplanes to run this Lab.)

If you double buffered *color.c* now, it wouldn't look very different because *color.c* is static. First add some motion to it, then double buffer it to improve the quality of the moving image.

Make a copy of *color.c* and call it *motion.c*. *motion.c* will draw a red ball that flies over a stationary blue rectangle. Edit *initialize* and *drawimage* so they look like the code on the next page.

```
initialize()
{
     int i;

     prefposition(0, 1023, 0, 767);
     winopen("motion.c");

/*-----------------------------------------
  Make color index 12 contain 200 units
  of red.
-----------------------------------------*/
     mapcolor(12, 200, 0, 0);
}
drawimage()
{
     int j;

     for(j=0; j<400; j=j+3)
     {
          color(BLACK);
          clear();

/*-----------------------------------------
  Draw a solid blue rectangle (color index 4)
  at this position.
-----------------------------------------*/
          color(4);
          rectfi(100,100, 250,400);

/*-----------------------------------------
  Draw a red ball (color index 12) at
  position j, j.
-----------------------------------------*/
          color(12);
          circfi(j, j, 20);
     }
}
```

Compile and run *motion.c*. You can eliminate all of this flashing by double buffering this program.

Make a copy of *motion.c* and call it *dbuf.c*. To double buffer *dbuf.c*, follow three steps:

1. Turn on double buffer mode with **doublebuffer**, and tell the system to put this mode routine into effect with **gconfig**.
2. Clear the front buffer to black.
3. Tell *dbuf.c* to swap buffers after it draws each image by using **swapbuffers**.

Edit *dbuf.c* so it looks like this:

```
#include "gl.h"
#include "device.h"

main()
{
     initialize();
     drawimage();
     while(TRUE) {
          checkinput();
          }
}
initialize()
{
     int i;

     prefposition(0, 1023, 0, 767);
     winopen("color.c");

     mapcolor(12, 200, 0, 0);
```

```
/*-----------------------------------------
  Declare double buffer mode, and tell the
  system to start it now.  This enables
  only the back buffer.
-----------------------------------------*/
     doublebuffer();
     gconfig();

/*-----------------------------------------
  Enable the front buffer and clear it to
  black.  Then disable it so the program will
  write to only the back buffer the first
  time it draws an image.
-----------------------------------------*/
     frontbuffer(TRUE);
     color(BLACK);
     clear();
     frontbuffer(FALSE);

}
drawimage()
{
     int j;

     for(j=0; j<400; j=j+3)
     {
          color(BLACK);
          clear();

          color(4);
          rectfi(100, 100, 250, 400);

          color(12);
          circfi(j, j, 20);
```

```
/*-------------------------------------------
  Swap buffers after both images have been
  drawn.
-------------------------------------------*/
          swapbuffers();
     }
}
checkinput()
{
     short val;

     switch(qread(&val)) {
          case REDRAW:
               reshapeviewport();
               break;
          }
}
```

Compile and run *dbuf* to see how much better it looks than *motion.*

Workshop

Look over *double3.c.* Because this program now has motion, you should double buffer it.

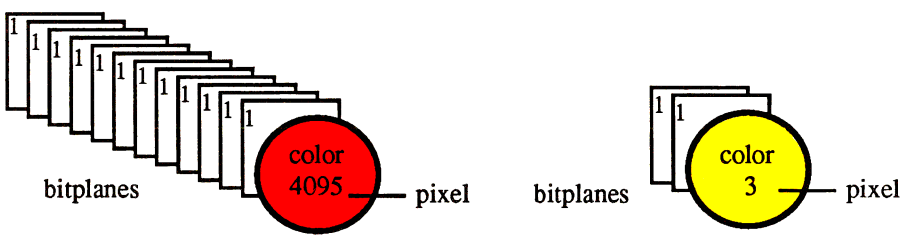
Writemasks

You can avoid a lot of time-consuming, complex redrawing if you use writemasks. To understand how writemasks work, you need to understand how the IRIS uses bitplanes to store information about color. For now, assume that you are using single buffer mode.

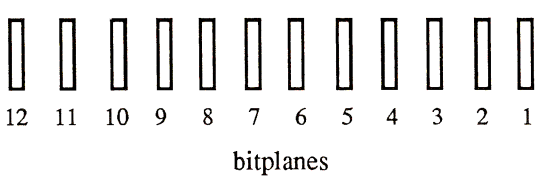
Bits, Bitplanes, and Binaries

Behind each pixel on your screen there are several bits that hold the color index value for the pixel. If you have 12 bitplanes in your IRIS, then there are 12 bits devoted to each pixel. Since you are dealing with bitplanes, it is helpful to think of color indices as binary numbers. Each digit corresponds to one bitplane.

For example, color number 4095 is expressed in binary as 111111111111. So, to display color number 4095, you need to have twelve bitplanes (one for each digit). If you want to display color 3 (11 in binary), you need only two bitplanes.



The term *first bitplane* corresponds to the bit that holds the right most digit (the first-order digit) in the binary.



For this discussion, assume that you have only three bitplanes. You'll learn the implications of having more bitplanes at the end of this section.

Every time you change the color of a pixel, you give the pixel a new color index value; that is, you change the values stored in the bits that keep track of that pixel. So, if the present value of the pixel is 000, the IRIS needs to change (write to) only the first two bitplanes to produce color 3 (011). However, if the present value is 4 (100), the IRIS must change all three bitplanes to produce color 3. If it changed only the first two bitplanes, you would see color 7 (111). Bitplane three would retain its value, while bitplanes one and two stored the new values.

By default, the IRIS writes to all bitplanes whenever you specify a new color for a pixel. This way, you don't see an unexpected color (like color 111, above). However, there may be times when you don't want to write to all bitplanes. The device you use is a writemask.

A writemask is a binary number that tells the IRIS which bitplanes it can and cannot write to. A zero (0) means don't write to this bitplane; a one (1) means do write to it. The default writemask has a 1 in each position, meaning write to all bitplanes. For this three-bitplane example, the default writemask is 7 (111). When you choose a new writemask, it replaces the default values.

A writemask of 2 (010) tells the IRIS that bitplanes one and three are protected; that is, that the IRIS cannot change the values that are stored in those bitplanes. It also tells the IRIS that bitplane two is enabled, which means it can write to it (change its value).

Here's what happens if you use a writemask of 2 (010) in the example above.

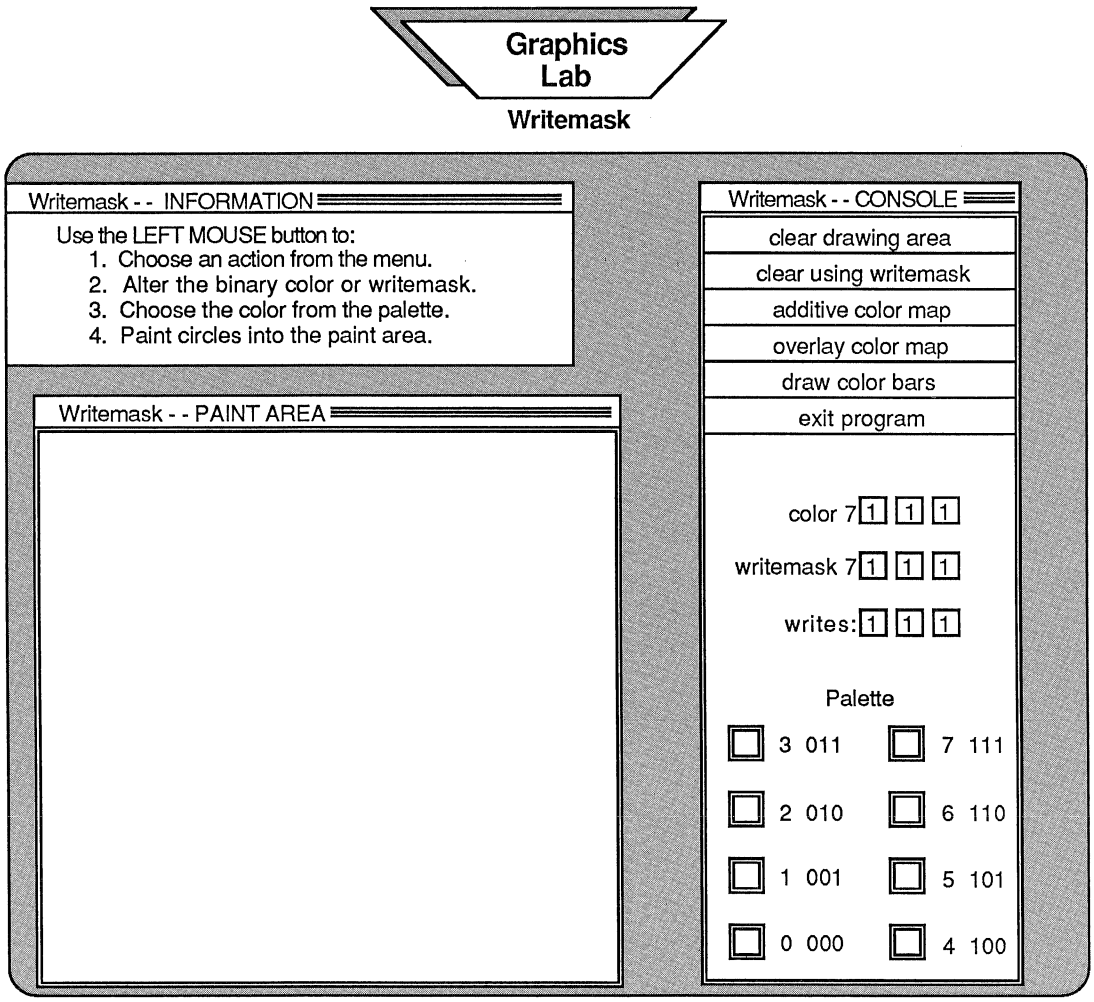
1. You want to display the color 3 (011).
2. The writemask is set to 2 (010).
3. The current value in the bitplanes is 4 (100).
4. The IRIS checks the current value and the writemask. To display color 3 it would have to change all three bitplanes. Bitplanes one and three are protected, so the IRIS can't change their values. Bitplane two is enabled, so the IRIS can change its value. The color it finally displays is color 6 (110).

desired new color	0 1 1
writemask	0 1 0
existing color	1 0 0
displayed color	1 1 0

— Please put these values in bitplanes 1 through 3.

— Sorry, bitplanes 1 and 3 are protected.

— I have to display the protected values in bitplanes 1 and 3, plus the new value in bitplane 2.



Run *writemask* to learn how the writemask affects the color that is displayed. Ignore the "additive color map" menu item for now. (Your IRIS must have at least 20 bitplanes to run this Lab.)

Why Use Writemasks?

It may seem that writemasks only make life difficult. The relationship between writemasks and color *is* a little tricky, but writemasks are extremely valuable.

For example, say you are writing a program where there is a constant background (like a baseball diamond), and you want to make an object (a ball) seem to move over the diamond. One way to do this is to redraw the entire picture several times with the ball in a slightly different position. But wouldn't it be great (and faster) if you needed only to redraw the ball in its new position, and never redraw the diamond? Here's where writemasks come in handy.

You use a writemask to protect the bitplanes that store the color indices of a stable image (the diamond). Then, you use the enabled bitplanes to store the color indices for another image (the ball). When the ball is displayed over the diamond, the enabled bitplanes (for the ball) change — not the protected ones (for the diamond). When the ball moves on to a new position, the color indices for the diamond at the ball's old position are still intact, so you don't have to redraw the diamond.

Here's how to set up a writemask for the baseball example. To make this example more realistic, assume that you have eight bitplanes.

1. You need to make a writemask to protect the bitplanes that store all of the colors that make up the diamond. The diamond has grass of color 8 (00001000), fences of color 9 (00001001), and an outfield of color 0 (00000000). Your writemask must protect bitplanes one through four. A writemask of 16 (00010000) will do the trick. There are several others you could choose, such as 240 (11110000), depending on how many enabled bitplanes you need. For now, use a writemask of 16.

2. For the ball, you must select a color index that can be stored in the fifth bitplane, since it is the only bitplane you can change. Assign the ball a color index of 16 (00010000).

3. Here's the tricky part. Remember, although you cannot write to the protected bitplanes, the values that are stored in them will affect the color of the ball. You assigned the ball the color index 16 (00010000), and you have a writemask of 16 (00010000) protecting the first four and last three bitplanes. What happens when the ball is displayed over the diamond (color 00001000)? The ball is displayed in color 24 (00011000). Then when it passes over the fence (color 00001001), it is displayed in color 25 (00011001). Finally, when the ball reaches the outfield (color 00000000), it is color 16 (00010000).

chosen color index for baseball	0 0 0 1 0 0 0 0
writemask	0 0 0 1 0 0 0 0
color indices for diamond	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1
all possible color indices for baseball	0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1 0 0 1

This is a problem, because baseballs don't generally change to three different colors like this. To make sure the ball stays the same color no matter where it is on the screen, you must:

1. calculate all possible colors that appear when you mix the ball's color index with the protected indices. In this case they are the colors whose indices are 16, 24, are 25 (see above).

2. load the color map so that all of the possible color indices for the ball (16, 24, and 25) point to the same RGB value, that is, the same color.

Index	RGB (color)
0	black
8	green
9	white
16	orange
24	orange
25	orange

The workshop below illustrates how this example works.

Workshop

Look over the code for *overlay4.c*. Notice that you turn off the writemask with the routine `writemask(0xfff);`.

Using a Writemask

Make a copy of *dbuf.c* and call it *write.c*. Since the blue rectangle never moves, you can draw it only once and protect it with a writemask. The black background is index number zero (000000), and the blue rectangle is index number four (000100). These colors use the first three bitplanes. A writemask of 8 (001000) will protect them.

Remember, the ball is color 12 (001100). It travels over both blue and black areas. If you protect the first three bitplanes after you draw the rectangle and the background, those bitplanes will contain either 000100 or 000000. When the ball travels over a blue area, the IRIS displays color 12 (001100). When it travels over a black area, the IRIS displays color 8 (001000).

ball	0 0 1 1 0 0
writemask	0 0 1 0 0 0
background blue rectangle	0 0 0 0 0 0 0 0 0 1 0 0
ball over background ball over rectangle	0 0 1 0 0 0 0 0 1 1 0 0

This means you must make color 8 the same as color 12 so the ball is always displayed in the same color. Edit *initialize* and *drawimage* so they look like this:

```
initialize()
{
     int i;

     prefposition(0, 1023, 0, 767);
     winopen("color.c");

     mapcolor(12, 200, 0, 0);

/*-----------------------------------------
  Make color index 8 the same as index 12.
-----------------------------------------*/
     mapcolor(8, 200, 0, 0);
```

```
    doublebuffer();
    gconfig();

    frontbuffer(TRUE);
    color(BLACK);
    clear();

    frontbuffer(FALSE);
}
drawimage()
{
    int j;

/*-----------------------------------------
  Draw the rectangle into both buffers.
-----------------------------------------*/
    frontbuffer(TRUE);
    color(4);
    rectfi(100, 100, 250, 400);
    frontbuffer(FALSE);

/*-----------------------------------------
  Protect bitplanes 1-3 with a writemask of 8.
-----------------------------------------*/
    writemask(8);

    for(j=0; j<400; j=j+3)
    {
        color(0);
        clear();

        color(12);
        circfi(j, j, 20);

        swapbuffers();
    }
```

```
/*-------------------------------------------
  Turn off the writemask.
-------------------------------------------*/
     writemask(0xfff);
}
```

Compile and run *write.c*. It looks just like *dbuf*. Try some different writemasks and see what happens.

Modes and the Window Manager

Color, double buffering, and writemasks affect the way you use the window manager, *mex*. Pop-up menus are like the baseball — they are another kind of overlay. When you use *mex*, the last two bitplanes are reserved for pop-up menus. This has two implications:

- You have fewer colors available for drawing your images.
- You can use only three different colors for the pop-up menus and the cursor.

Also, when you use double buffer mode, the window manager needs four bitplanes (two for each buffer). This further reduces the number of bitplanes in which you can store information about your images.

Summary

- There are 256 possible intensities for each of the three basic colors (red, green, and blue).
- The IRIS color map stores 24-bit colors so your bitplanes don't have to.
- Programs that are running simultaneously share the same color map.
- Double buffering smooths motion, and solves the problem of half drawn images. However, it also cuts the number of bitplanes available for storing image information in half.
- Writemasks help you avoid extra redrawing of stationary images.
- *mex* takes up two full bitplanes in single buffer mode, and four in double buffer mode.

5 Interacting with the User

At the beginning of the tutorial you learned that interactivity can be an important feature of graphics programs. A user gives input to the program, and the program uses it to create a new image. The user can interact through a number of different devices:

- the keyboard
- the mouse
- a light pen
- a dial and button box
- a digitizer tablet

The input can be of three types:

- An integer value. For example, the mouse inputs two values: the horizontal and vertical positioning of the cursor. All devices that input integer values are called *valuators*.
- A Boolean (true/false) value. For example, the mouse buttons give Boolean input — they are either down or up (true or false). All devices that input Boolean values are called *buttons*.
- An ASCII character. This type of input comes from the keyboard. Each key has a different value.

Polling and Queueing

Your program can handle input from the user in two ways: by *polling* and *queueing*. To poll an input device, you include a routine in your program that checks the current state of the device. For example, your program may be running and suddenly you want to know if the right mouse button is being pressed. So, you include a routine that returns the value of the right mouse button (TRUE = pressed; FALSE = not pressed). Or, you may want to know where the cursor is located, so you include a routine that returns the *x* and *y* coordinates of the cursor.

You should always be aware of one important fact: the system ignores any input that the user gives before the program polls. The user may have been moving the cursor and clicking buttons for minutes, but the only location that the program knows about is the one at the time of polling.

Queueing, on the other hand, saves input from the user in an event queue until your program is ready to deal with it. To use queueing in your programs, you first set up an event queue by telling the program which input devices to recognize. These input devices are now called *queued devices*. Whenever the user changes the state of a queued device, an entry is made in the event queue. For example, if you queue the keyboard, an entry appears in the event queue whenever a key is pressed. When the program is ready, it reads the event queue and processes the events that have occurred. In this case, the program reads all the keys that have been pressed, and performs a different function for each key.

To summarize, there are three types of input: valuators, buttons, and the keyboard. And, there are two ways your program can handle input: polling and queueing. Most of this chapter concentrates on queueing, but the workshop below shows a good example of polling.

Workshop

Look over *poll5*. It uses the `getbutton` routine to determine where the cursor is.

The Program Template Revisited

Take a look at the template again. You need to change it so it can handle input from the user. The major difference is that *checkinput* will now be called *processinput*. All of the changes are in this function (except for the one line in *main* that calls it). Make a copy of *template.c* and call it *template2.c*. Edit it so it looks like this code.

```
/*------------------------------------------------*/
#include "gl.h"
#include "device.h"
/*------------------------------------------------*/
main()
{
     initialize();
     drawimage();
     while(TRUE)
          processinput();
}
/*------------------------------------------------*/
initialize()
{
     prefposition(0, 1023, 0, 767);
     winopen("template2");
}
/*------------------------------------------------*/
processinput()
{
     short val;

     switch(qread(&val)) {
          case REDRAW:
               reshapeviewport();
               drawimage();
               break;
     }
}
```

```
/*-----------------------------------------------*/
drawimage()
{
     color(BLACK);
     clear();
}
/*-----------------------------------------------*/
```

Remember, the basic algorithm is a loop that:

1. draws an image and then processes input
2. draws a new image based on that input and then processes new input

You've already learned about what you can do with the *drawimage* function. Now take a look at *processinput.*

Right now, *processinput* does only one thing: it checks to see if the window manager says the image should be redrawn. The window has to be redrawn if you move or reshape it, or if another window covers it up. The first window may have to be redrawn if you move the other window.

Any program that runs under the window manager must look at the event queue. The window manager uses the queue to tell the program when to redraw a window. It does this by putting a special token (REDRAW) in the queue. If your program doesn't look at its event queue, nothing happens. In this case, if you moved your window, it wouldn't be redrawn in the new position. So, at the very least, *processinput* needs to check the event queue for the REDRAW token.

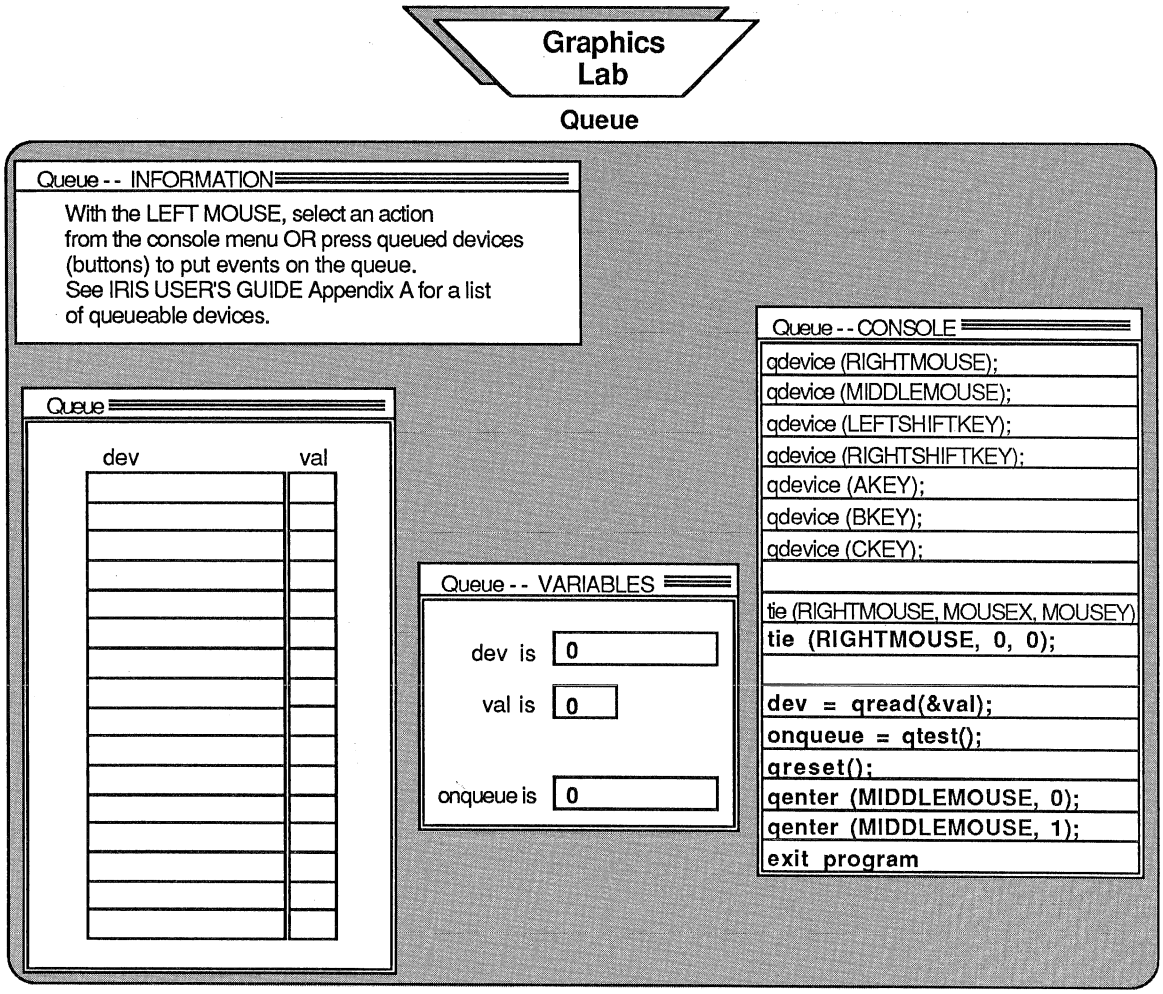
The key routine in *processinput* is **qread.** When you call **qread**, it returns the device number of the first entry in the event queue. (Each input device is assigned a number.) After **qread** reads an entry, it removes the entry from the queue. However, if there is nothing in the event queue, the program waits in the **qread** routine. This means your program won't do a thing until something is put into the queue.

This is how the REDRAW token fits in when you run your program.

1. *drawimage* draws an image in the window.
2. Your program calls *processinput*. The program waits there until the window manager says the window needs to be redrawn.
3. If you move or reshape the window, an entry is made in the event queue.
4. `qread` returns a device number, which is equal to the value of REDRAW.
5. This sends the program into the REDRAW case. The window is redrawn.
6. The program executes *drawimage*. A new image is drawn in the new window, the program calls *processinput*, and waits for another notice to redraw.

Another important element is the variable *&val*. Every entry in the event queue has two parts: the device number which is returned by `qread`, and some value that is associated with the device. This associated value is stored in *val*, a short integer. The & sign tells the program to look at the memory address of *val* because this is where the value part of the entry is stored.

Note that the template doesn't seem to do anything with the value from the REDRAW entry. This is because it is a special token unique to the window manager. Even though the program doesn't use *val* for REDRAW, you still need to declare the variable. The *val* of other types of entries is very important. For example, if you queue the keyboard, *val* tells the program which key was pressed.



Run *queue*. Use the left mouse button to queue some devices that are in the first menu. Then, use these devices and watch them appear in the queue. Finally, use the left mouse button to do a `qread`. (Your IRIS must have at least 12 bitplanes to run this Lab.)

Your First Interactive Program

This is the first program you can talk to *after* you compile it. You start by queueing something in addition to the REDRAW token. Change your program so that when you push the middle mouse button, a red circle is drawn in the middle of the screen. Make a copy of *template2.c* and call it *circleio.c*.

1. First, you need to queue up the middle mouse button. **`qdevice`** does the job. It takes the name or number of an input device as its argument. When the device changes state, an entry for that device is put in the event queue. You need to do this only once for the program, so it goes in the *initialize* function.

2. Next, you need to check the queue for middle mouse entries. Add another case to *processinput*. Whenever the program reads a middle mouse button event from the queue, it sets the global variable *drawcircle* to 1, and then calls *drawimage* again.

3. To use the *drawcircle* variable, you must declare it at the beginning of the program, and initialize it to 0 so the circle won't be drawn until the middle mouse button is pressed.

4. Finally, you need to change *drawimage* so that whenever *drawcircle* is equal to 1, a circle is indeed drawn.

Edit *circleio.c* so it looks like the code below.

```
#include "gl.h"
#include "device.h"

/*-------------------------------------------------
Declare a variable called drawcircle.
-------------------------------------------------*/
short drawcircle;
```

```
main()
{
     initialize();
     drawimage();
     while(TRUE) {
          processinput();
     }
}
initialize()
{
     prefposition(0, 1023, 0, 767);
     winopen("circleio");

/*-------------------------------------------------
Queue the middle mouse button.
-------------------------------------------------*/
     qdevice(MIDDLEMOUSE);

/*-------------------------------------------------
Set drawcircle to 0 so nothing is drawn until the
middle mouse button is pressed.
-------------------------------------------------*/
     drawcircle = 0;
}
processinput()
{
     short val;

     switch(qread(&val)) {
          case REDRAW:
               reshapeviewport();
               drawimage();
               break;
```

```
/*-------------------------------------------------
When the middle mouse is pressed, set drawcircle
to 1 and go to drawimage.
-------------------------------------------------*/
          case MIDDLEMOUSE:
               drawcircle = 1;
               drawimage();
               break;
     }
}
drawimage()
{
     color(BLACK);
     clear();

/*-------------------------------------------------
If drawcircle = 1, the button has been pressed, so
draw a red circle.
-------------------------------------------------*/
     if (drawcircle == 1){
          color(RED);
          circfi(100, 100, 100);
     }
}
```

Try this program. Once you've got the idea, you can make your program even better.

Improvement #1

What happens when you run this program?

1. The *main* function (as in every C program) is executed first.
2. It calls *initialize*, which makes the window take up the whole screen, sets *drawcircle* to 0, and queues the middle mouse button.

3. It calls *drawimage*, which clears the screen to black. Since *drawcircle* is 0, *drawimage* does nothing else.

4. The program goes into a loop that doesn't end until you exit from the window by selecting "kill" from the *mex* menu.

5. *processinput* is within this loop. It declares *val* as a short integer, and then it waits for an event to appear in the queue. (Don't worry about reshaping or moving the window now — assume that no redraws will show up for a while.)

So nothing happens until the middle mouse button is pressed. But, when the user presses the middle mouse button, the excitement begins.

1. `qread` stores the entry in *val* and deletes it from the event queue.

2. `qread` returns the value of the device, that is, the middle mouse button device number.

3. The program executes the MIDDLEMOUSE case.

4. *drawcircle* is set to 1 and the program calls *drawimage*.

5. *drawimage* clears the screen to black and, since *drawcircle* now equals 1, *drawimage* draws a red circle on the screen.

6. Then the program returns to *processinput* and waits for another event.

Now that this event has been processed, what's the next event going to be? The last event was the middle mouse button going down, so the next event will probably be the mouse button going up. The program will call *drawimage* again, clearing the screen and drawing the red circle. This will make the screen image flash. How can you avoid this?

You can't change the fact that releasing the mouse button is an event. Once the middle button is queued, an event is put in the queue when this button goes up. What you really want to do is draw the circle when the mouse button goes down, and ignore the mouse button when it goes up. **qread** reads the release of the button, and calls the MIDDLEMOUSE case. This is where you can check whether the button is up or down, and can choose to ignore the event or not.

Remember, when an event occurs, the event queue stores both the device number and some associated value. The device numbers for the mouse going up and for the mouse going down are the same; the associated values are different. Going down is 1 and coming up is 0. So, you have to call *drawimage* only if *val* is 1. Edit your code:

```
processinput()
{
     short val;

     switch(qread(&val)) {
          case REDRAW:
               reshapeviewport();
               drawimage();
               break;
          case MIDDLEMOUSE:
               if (val == 1){
                    drawcircle = 1;
                    drawimage();
               }
               break;
     }
}
```

Improvement #2

Now the problem is that the program is kind of boring. Once you've pressed the middle mouse button and the circle has appeared, there's nothing left to do. And the image still flashes when you press the button because the circle is redrawn. First make your program smarter, then prettier.

Make the program erase the circle when the user presses the left mouse button.

1. Queue the left mouse button in the *initialize* function.
2. Add the left mouse button case so that *drawcircle* is set to 0 when the left mouse button is pressed. Call *drawimage* to clear the screen.

```
processinput()
{
     short val;

     switch(qread(&val)) {
          case REDRAW:
               reshapeviewport();
               drawimage();
               break;
          case MIDDLEMOUSE:
               if (val == 1) {
                    drawcircle = 1;
                    drawimage();
               }
               break;
          case LEFTMOUSE:
               if (val == 1) {
                    drawcircle = 0;
                    drawimage();
               }
               break;
     }
}
```

Now the program is more sophisticated. When you press the middle mouse button, a red circle appears. When you press the left mouse button, the circle disappears. What could be better? Well...

Improvement #3

This program is still kind of ugly — it flashes when you draw the circle. That's because you still see the IRIS drawing the object. You can use double buffering to solve this problem.

1. Add some routines to *initialize* that turn on double buffering and clear both buffers to black.

```
initialize()
{
     prefposition(0, 1023, 0, 767);
     winopen("circleio");

     doublebuffer();
     gconfig();

/*-------------------------------------------
  Clear both buffers to black
-------------------------------------------*/
     frontbuffer(TRUE);
     color(BLACK);
     clear();
     frontbuffer(FALSE);

     qdevice(MIDDLEMOUSE);
     qdevice(LEFTMOUSE);

     drawcircle = 0;
}
```

2. Now add a swap at the right point.

```
drawimage()
{
     color(BLACK);
     clear();

     if (drawcircle == 1) {
          color(RED);
          circfi(100, 100, 100);
     }
     swapbuffers();
}
```

Improvement #4

To take interactivity a step farther, make the circle appear in the same location as the cursor. To do this, you need to poll the location of the cursor when you're ready to draw the circle. The *x* and *y* coordinates of the cursor location are valuators. Their names are MOUSEX and MOUSEY.

The routine for polling a valuator is **`getvaluator`**. Set the *x* argument of the **`circfi`** routine to MOUSEX, and the *y* argument to MOUSEY.

```
drawimage()
{
     int x,y;

     color(BLACK);
     clear();
```

```
        if (drawcircle == 1) {
            color(BLUE);
            x = getvaluator(MOUSEX);
            y = getvaluator(MOUSEY);
            circfi(x, y, 100);
        }
        swapbuffers();
}
```

Compile and run this program. It's come a long way.

Workshop

Take a look at *aim6*. The batter hits the ball to the spot where you click the mouse button.

Running More than One Double Buffered Program

The problem you face when running more than one double buffered program is that all programs have to swap buffers at the same time. This is because all the programs share the same image memory. When the two buffers in that memory are swapped, the buffers for all your programs are swapped, whether you want this to happen or not.

The system prevents the disaster of unexpected swaps by waiting until all double buffered programs are ready to swap. This means that whenever a double buffered program calls a `swapbuffers` routine, it waits there until all the double buffered programs are ready to swap. You can see the problem if you run *circleio* and leave it idle for a few minutes while you run another program. The program *circleio* waits for input at `qread` — not at `swapbuffers` waiting for a swap. As a result, all the double buffered programs would just hang — some of them on `swapbuffers`, and at least this one on `qread`.

So, you need to make sure double buffered programs always wait at the `swapbuffers` routine. Take a look at the following version of the *circleio* program. It introduces a new routine called in *main* called `qtest`. `qtest` looks at the event queue and returns the device number of the first entry, just like `qread`. However, it differs from `qread` in two important ways:

1. It returns 0 if the queue is empty. It doesn't wait for an entry in the event queue.
2. It doesn't remove any entries from the event queue. It only checks to see if any entries are there.

The basic change to your program is that it must continuously check the queue with `qtest`. If the queue is empty, it calls *drawimage* which calls `swapbuffers`. If there is an entry in the queue, it does a `qread` and processes the input as usual. Then it continues checking for input and doing swaps.

Copy *circleio.c* and name the copy *circledb.c*. Then edit it to look like this code.

```
#include "gl.h"
#include "device.h"

short drawcircle;
main()
{
     initialize();
     drawimage();
     while(TRUE) {
/*-----------------------------------------------
  Check the queue.  If it's empty, go to
  drawimage.  It there is an event, process it.
-----------------------------------------------*/
          while (!qtest())
               drawimage();
          processinput();
     }
}
initialize()
{
     prefposition(0, 1023, 0, 767);
     winopen("circledb");

     doublebuffer();
     gconfig();
     frontbuffer(TRUE);
     color(BLACK);
     clear();
     frontbuffer(FALSE);

     qdevice(MIDDLEMOUSE);
     qdevice(LEFTMOUSE);

     drawcircle = 0;
}
```

```
processinput()
{
     short val;

     switch(qread(&val)) {
          case REDRAW:
               reshapeviewport();
               drawimage();
               break;
          case MIDDLEMOUSE:
               if (val == 1) {
                    drawcircle = 1;
                    drawimage();
               }
               break;
          case LEFTMOUSE:
               if (val == 1) {
                    drawcircle = 0;
                    drawimage();
               }
               break;
     }
}
drawimage()
{
     int x, y;

     color(BLACK);
     clear();

     if (drawcircle == 1) {
          color(RED);
          x = getvaluator(MOUSEX);
          y = getvaluator(MOUSEY);
          circfi(x, y, 100);
     }
     swapbuffers();
}
```

Look at the Graphics Lab *queue* again to see how `qtest` works.

Workshop

Look over the code for *queue7.c* to see another queue in action. Also read through the code for *menu8.c*. It shows you how to add a pop-up menu to the baseball example. This tutorial does not cover pop-up menus. If you are interested in learning about them, see *Using mex, the IRIS Window Manager* in the *IRIS User's Guide*.

Summary

- There are three types of input: integer, boolean, or ASCII.
- Whether you choose to use polling or queueing depends on your particular application.
- You must always have the REDRAW token in the queue when your program runs under *mex*.
- At any given time, every device has a device number and a value.
- When a button goes down and up, the IRIS interprets it as two different events, and makes two entries in the queue.
- When you write a double buffered program that runs at the same time as another double buffered program, you must place the `swapbuffers` routine strategically, and use `qtest`.

6 Creating and Manipulating 3D Models

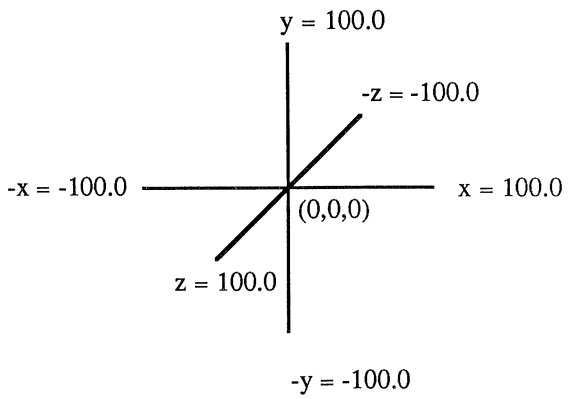
This chapter leads you through the transition from drawing 2D images to building 3D models. Then, you'll learn how to move these models around in space.

Building 3D Models

You build 3D models with the same routines you use to draw 2D images. You actually construct the object in three dimensions and let the system figure out how to display it on a 2D screen.

So far it's worked well to use the 2D screen coordinate system (1024x768) to specify the locations of your 2D objects. But now that you're making 3D models, you need to use a 3D coordinate system. You can create this system with a Graphics Library routine.

Since you create this system yourself, you can make it very convenient to work with — for example, you can place its origin exactly in the middle of a window so whatever you draw is centered. You can also make the axes all the same length.

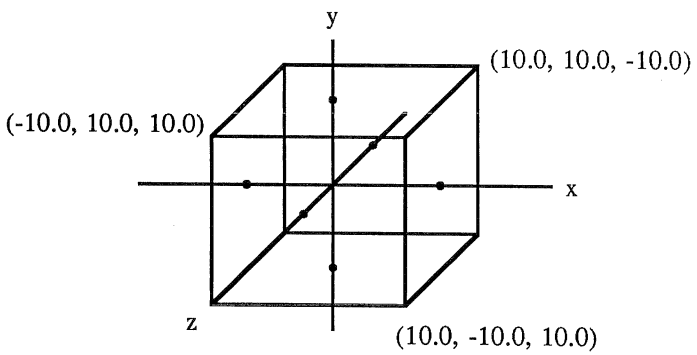


Make a copy of *template2.c* and call it *cube3d.c*. Create this coordinate system in *cube3d.c* by adding this code as the last line of the *initialize* function.

```
ortho(-100.0, 100.0, -100.0, 100.0, -100.0, 100.0);
```

Now you can create a model within this system. A cube is a classic 3D object for a start.

Your first decision is where to place the cube. All sides of the cube are equal, so it's easiest to pick vertices that differ only in their signs. For example, one vertex of a cube could be (10.0, 10.0, 10.0), while another could be (10.0, 10.0, -10.0). To use equal values like these, the cube must be centered at the origin.



The next point to consider is whether your cube will be wireframe or solid. This affects which drawing routines you use.

Building with move and draw

To build a wireframe model, you can use **move** and **draw**. They work the same way as in 2D, except now you use the 3D versions of these routines. Use the 3D/floating point versions of **move** and **draw** to model a cube about the origin. Create a new function at the end of *cube3d.c* called *drawcube*:

```
drawcube()
{
     move(-10.0, -10.0, -10.0);
     draw(10.0, -10.0, -10.0);
     draw(10.0, 10.0, -10.0);
     draw(-10.0, 10.0, -10.0);
     draw(-10.0, -10.0, -10.0);
     draw(-10.0, -10.0, 10.0);
     draw(10.0, -10.0, 10.0);
     draw(10.0, 10.0, 10.0);
     draw(-10.0, 10.0, 10.0);
     draw(-10.0, -10.0, 10.0);
     move(-10.0, 10.0, -10.0);
     draw(-10.0, 10.0, 10.0);
     move(10.0, 10.0, -10.0);
     draw(10.0, 10.0, 10.0);
     move(10.0, -10.0, 10.0);
     draw(10.0, -10.0, -10.0);
}
```

Now you can call *drawcube* from *drawimage*:

```
drawimage()
{
     color(BLACK);
     clear();

     color(RED);
     drawcube();

     swapbuffers();
}
```

Compile and run your program. You see a red square because the front edges of the cube obscure all the other edges. (Later in this chapter, you'll learn how to turn it around to get a better view.)

threed9.c uses 3D versions of the drawing commands, and *coord10.c* sets up a 3D coordinate system.

The problem with using **move** and **draw** is that after you build a model, you may decide that you want to color one side, or color all sides to make it look solid. The IRIS considers each line of a wireframe built with **move** and **draw** to be a separate object — it doesn't realize that your cube is a cube and that you should be able to color its sides. But, there is an alternative method that solves this problem.

Building with poly and polf

Another way to model a cube is to create six square polygons that make up the six faces of the cube. If you use **poly** you get a wireframe; if you use **polf** you get a solid model. If you start out designing a wireframe cube, then change your mind when it's done, change **poly** to **polf**. Here is part of a *drawpoly* function that draws a solid cube. (You don't need to type this in — it's in the revised *template2.c* file called *poly3D.c*.)

```
drawpoly()
{
/*----------------------------------------
  Set up a 3D array.
----------------------------------------*/
     {
     Coord parray [4][3];

/*----------------------------------------
  Build the back face of the cube.
----------------------------------------*/
     parray [0][0] = -10.0;
     parray [0][1] = -10.0;
     parray [0][2] = -10.0;
```

```
    parray [1][0] = 10.0;
    parray [1][1] = -10.0;
    parray [1][2] = -10.0;
    parray [2][0] = 10.0;
    parray [2][1] = 10.0;
    parray [2][2] = -10.0;
    parray [3][0] = -10.0;
    parray [3][1] = 10.0;
    parray [3][2] = -10.0;

/*-----------------------------------------
  Draw the back face.
-----------------------------------------*/
    polf(4, parray);
    }

/*-----------------------------------------
  Set up another 3D array.
-----------------------------------------*/
    {
    Coord parray [4][3];

/*-----------------------------------------
  Build the bottom face.
-----------------------------------------*/
    parray [0][0] = -10.0;
    parray [0][1] = -10.0;
    parray [0][2] = -10.0;
    parray [1][0] = -10.0;
    parray [1][1] = -10.0;
    parray [1][2] = 10.0;
    parray [2][0] = 10.0;
    parray [2][1] = -10.0;
    parray [2][2] = 10.0;
    parray [3][0] = 10.0;
    parray [3][1] = -10.0;
    parray [3][2] = -10.0;
```

```
/*---------------------------------------
  Draw the bottom face.
---------------------------------------*/
     polf(4, parray);
     }

/*---------------------------------------
  Continue this way to build and draw the
  front, top, left, and right faces.
---------------------------------------*/
}
```

If you compile and run this program, you see a square. The next section teaches you how to turn the cube around to get a better view.

Checking Out All the Angles

Now that you have described a 3D object to the IRIS, you can tell it to transform the object in several interesting ways. You use three routines called "modeling transformations": `rotate`, `translate`, and `scale`.

Rotate

`rotate` rotates your model about the *x*, *y*, or *z* axis. If you call `rotate` in the *initialize* function, everything you draw later is rotated. `rotate` takes two arguments: the number of degrees to rotate the object (in tenths of degrees), and the axis about which to rotate it (*x*, *y*, or *z*).

To rotate the cube 40 degrees around the *y* axis, change the *initialize* function in *cube3d.c*. Immediately after `ortho`, add this line:

```
rotate(400, 'y');
```

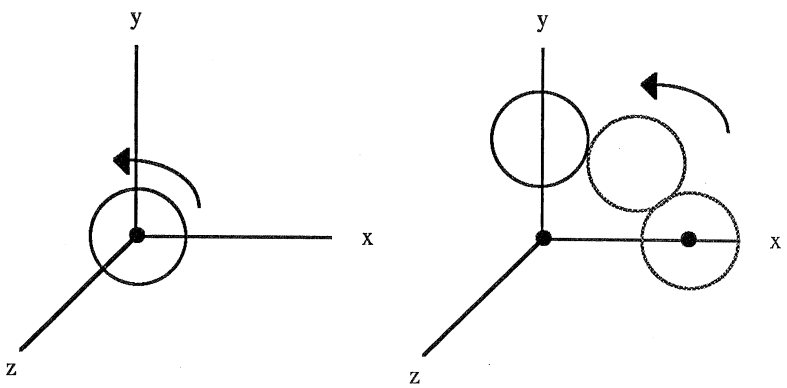
You still don't see anything that really looks like a cube. This is because you only rotated it in one dimension. Add a rotation about the *x* axis. Right after the first rotation, add this line:

```
rotate(450, 'x');
```

Finally, it looks like a 3D cube.

It is important to remember that objects rotate relative to the *x*, *y* and *z* axes. The position of an object relative to its coordinate system affects the way the object rotates.

For example, say you have a circle centered at the origin, and you want to rotate it about the *z* axis. The motion you see is like a wheel spinning about a hub. However, if the circle is not centered at the origin, the motion looks more like a planet orbiting around its sun.



Scale

`scale` makes all or part of your model larger or smaller. It uses relative values; that is, it makes an object a certain number of times larger or smaller than its current size. `scale` takes three arguments: the number of times larger or smaller to make the object in the *x*, *y*, and *z* directions. To make the cube twice as big, immediately after the last `rotate` routine, add this line:

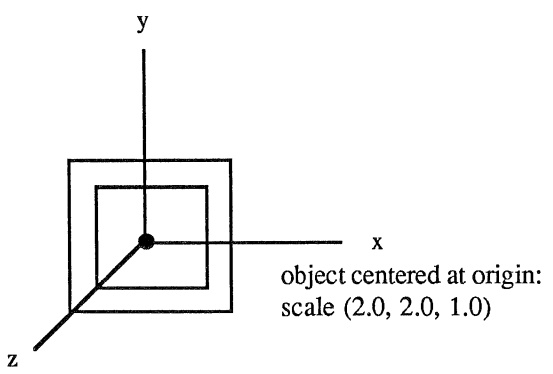
```
scale(2.0, 2.0, 2.0);
```

Never use zero (0) as an argument to `scale` — nothing will appear on the screen.

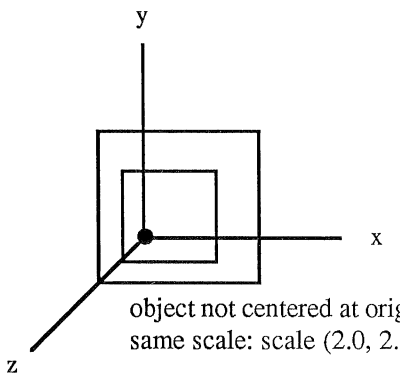
To stretch the cube in the *x* direction, remove the first `scale` routine, and replace it with this:

```
scale(4.0, 1.0, 1.0);
```

Like `rotate`, `scale` affects your objects differently depending on where they are located relative to the origin. For example, a square centered at the origin grows or shrinks evenly in every direction. If a square is not centered at the origin, all vertices grow or shrink away from the origin proportionally.

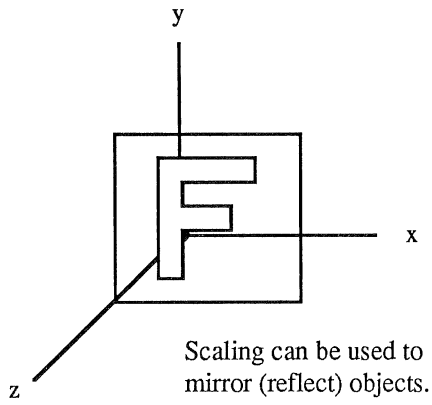


object centered at origin:
scale (2.0, 2.0, 1.0)

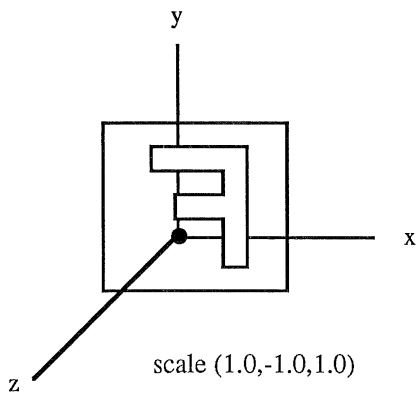


object not centered at origin,
same scale: scale (2.0, 2.0, 1.0)

Another interesting way to use `scale` is to create a mirror image of your object by using negative scaling values. In the figure below, the object is mirrored across the *y* axis.



Scaling can be used to
mirror (reflect) objects.



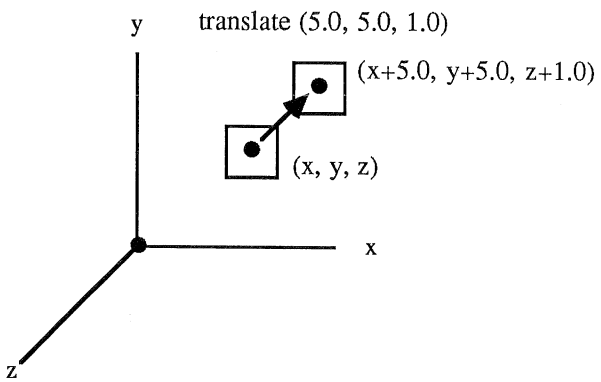
scale (1.0,-1.0,1.0)

Translate

translate moves your model around in the space (coordinate system) you defined with **ortho**. Like **scale**, it uses relative values, so it moves an object a certain distance from its current position. **translate** takes three arguments: the amount of distance to move the object in the *x*, *y*, and *z* directions. To move your cube from the center of the window to a new position, after the other modeling routines, add this line:

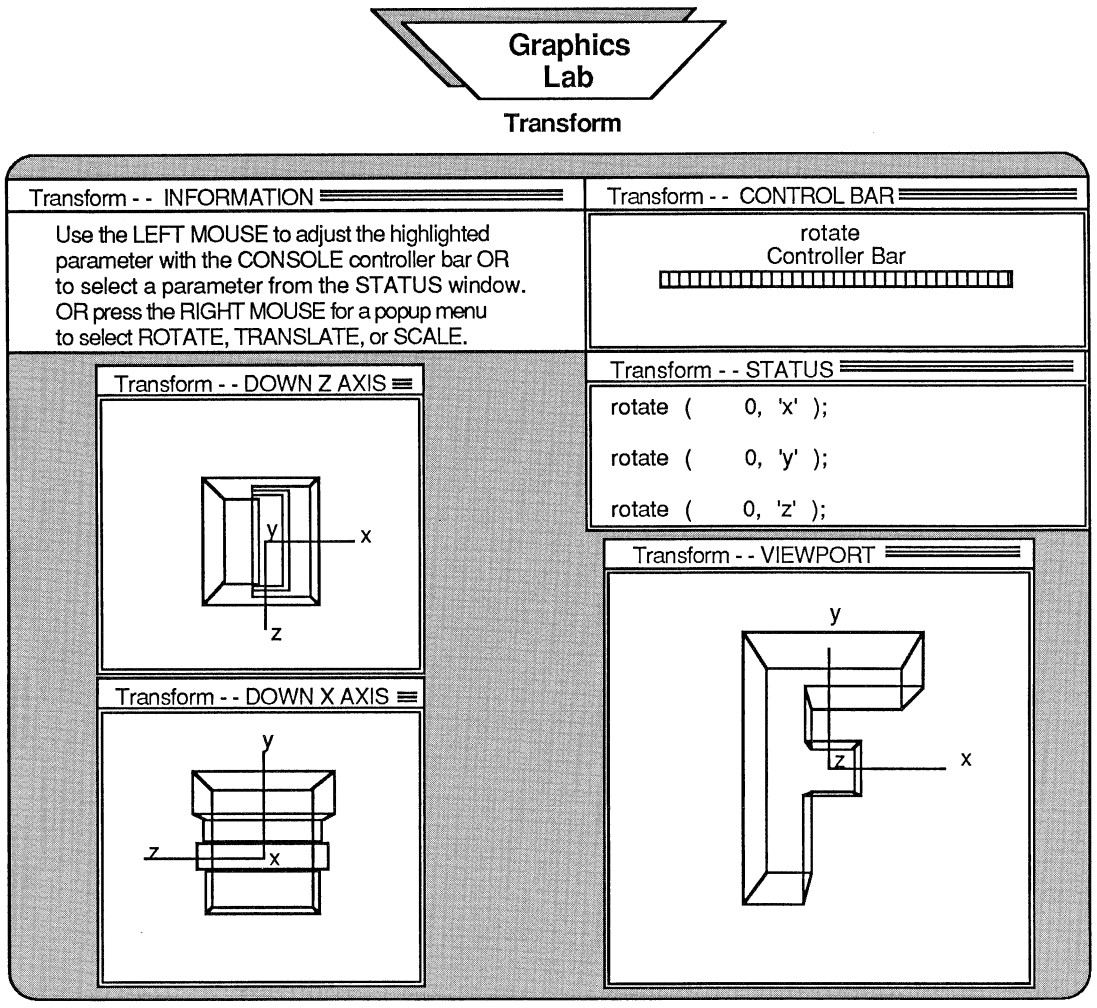
```
translate(30.0, 10.0, 0.0);
```

Now the cube is 30 units to the right and ten units above its former position.



Workshop

translate11.c moves the ball using modeling transformations.



Run *transform* to see these transformations in action. You can specify different modeling transformations and vary their parameters to see how they affect the image. (Your IRIS must have at least 12 bitplanes to run this Lab.)

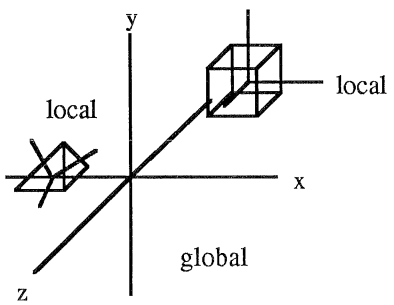
Go back to *cube3d.c*. The last change you made was to add a `translate` routine. What would happen if you added a `rotate` right now? Would it spin like a wheel, or orbit like a planet?

It would spin like a wheel, because when you translate an object, you also translate the object's coordinate system.

Transforming Coordinate Systems

When you build a model, you specify its vertices relative to a coordinate system. No matter what you do to the model, it always behaves in terms of its initial position in the coordinate system. For example, a cube that is built with its center at the origin always spins when you call `rotate`, and always grows or shrinks evenly when you call `scale`, regardless of how many times you call `translate`.

You can think of the coordinate system that you set up at the beginning of this chapter as a global system. Within this system you define objects. Once an object is defined, it retains its own local coordinate system as it moves around within the global coordinate system.



It's possible to have several objects, each with their own coordinate systems, floating around in this global system.

In the examples above, modeling transformation routines transform all objects that the program draws. But what if you want to draw five objects, four that are rotated by 30 degrees, and one that is not rotated but is scaled by a factor of six? You must call modeling transformations that affect local coordinate systems. However, the IRIS thinks that all transformation calls apply to the global coordinate system. To solve this problem, you need to understand how the IRIS performs transformations.

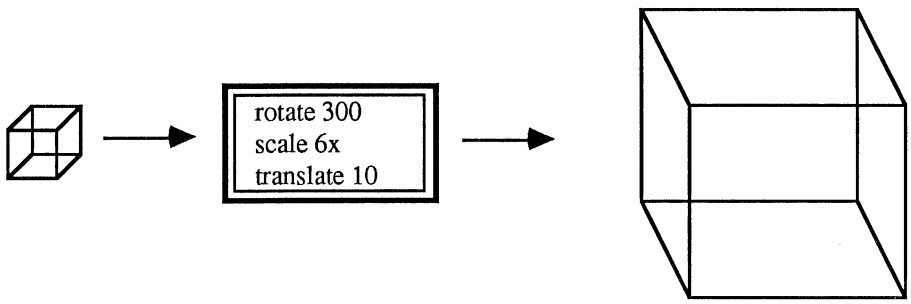
Manipulating the IRIS Transformation Matrix

The IRIS stores transformation information in 4x4 matrices. It can hold up to 32 individual transformation matrices in its matrix stack. The matrix at the top of the stack is called the current matrix.

When you call a modeling transformation, you are really telling the IRIS:

1. Store the transformation information in the current matrix.
2. Multiply every point of every object by the matrix.
3. Use these new values to draw the objects.

Modeling transformations are cumulative. This means that only the first transformation you call starts with a "fresh" matrix (a matrix that currently performs no modeling transformations). The first transformation call changes the values in the current matrix. When you make a second call, it changes the values again, so the current matrix makes both the first and second transformations happen in one multiplication. This process can go on and on, resulting in a matrix which, in one multiplication, calculates the points that result from several transformations.

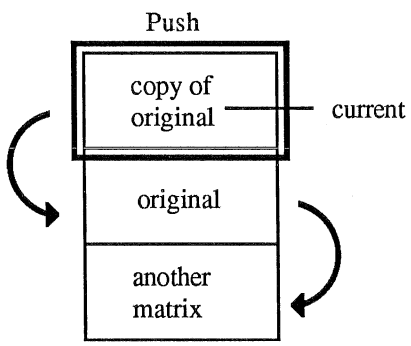


However, the more modeling transformations you call, the easier it is to loose your bearings within the global system. If the current matrix contains a 30 degree rotation, a 40 unit translation, a scaling factor of six, and a 48 degree rotation, it's difficult to anticipate where your objects will be displayed, and what they will look like. Remember, every object is multiplied by the current matrix.

The Graphics Library contains some matrix manipulation routines that save you from:

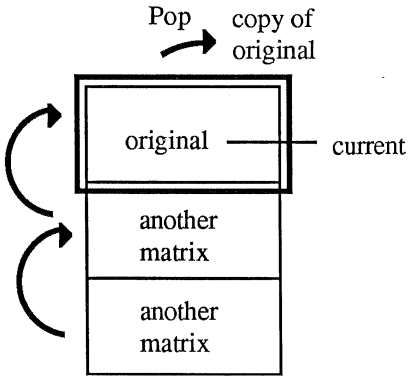
- keeping track of a current matrix that changes with every transformation call
- having all transformations apply to all objects that are drawn

The **pushmatrix** routine makes a copy of the current matrix, moves the current matrix and any other matrices that are below it down one position on the matrix stack, and puts the copy on top of the stack. Now this copy is the current matrix. You can change it as much as you want, and always know that you can restore the original at any time.



Matrix Stack

The **popmatrix** routine permanently deletes the current matrix and pops the matrix that was stored immediately below it into the current matrix position.



Matrix Stack

These routines are fast and efficient, so you can use them whenever convenient without sacrificing performance.

Use **pushmatrix** and **popmatrix** to solve the earlier problem of making a local transformation. The file called *matrix.c* in */usr/people/tutorial/c.graphics/explore* draws five objects without any modeling transformations. Compile and run it to see what it draws.

Make a copy of *matrix* and call it *mymatrix.c*. Edit this new file so it draws four objects that are rotated, and one that is not rotated but is scaled.

1. You haven't called any modeling transformations yet, so the current matrix doesn't contain modeling transformation information. You usually want to preserve this fresh matrix so you can recall it if needed. Use **pushmatrix** to make a copy of the original, push the original down one position, and put the copy in the top (current) position. Add this to *drawobjects* as the first line:

```
pushmatrix();
```

2. You want the first four objects to be multiplied by a matrix that will rotate them 30 degrees. So, call **rotate** to change the current matrix. After **pushmatrix**, add this line:

```
rotate (300, 'z');
```

3. If you compiled the program now, all five objects would be rotated. To prevent this, let the program draw the first four objects using the rotation matrix. Then, before it draws the fifth object, eliminate the rotation matrix, and restore the original. Copy the original, push it down on the stack, and add a **scale** to the copy. Edit *drawfive* so it looks like this.

```
drawfive()
{
/*-----------------------------------------
Delete the rotation matrix and restore
the original matrix.
-----------------------------------------*/
     popmatrix();

/*-----------------------------------------
Copy the original (current) matrix,
push it down one position on the stack, and put
the copy in the current position. The copy
is now current.
-----------------------------------------*/
     pushmatrix();

/*-----------------------------------------
Scale the object.
-----------------------------------------*/
     scale (6.0, 6.0, 6.0);
     rectf(-10.0, -10.0, 10.0, 10.0);
}
```

4. Finally, when all of the objects have been drawn, restore the original matrix. As the last line of *drawobjects*, add this:

```
popmatrix();
```

Compile and run your program, and compare it to the original version.

It may seem from this example that matrix routines only control which objects are transformed. However, they also provide another service: they prevent the objects from moving around each time they are redrawn.

If you call a modeling transformation within a function that the program calls more than once (like *drawimage*, *drawobjects*, or *drawfive*), each time it calls the function, the transformation is incorporated into the current matrix. This means that if you put a **`rotate`** in *drawimage*, the program initially draws an object with the specified rotation. But each time it redraws the object, the program rotates the object again. The end result is an object that is rotating in real time.

Models, Motion, and Matrices

You can use all the techniques you learned in this chapter to write an animated program. This program makes a ball that rolls towards you. The ball starts in the distance, and ends up right in front of you. Make a copy of *template2.c* and call it *roll.c*.

Building the Ball

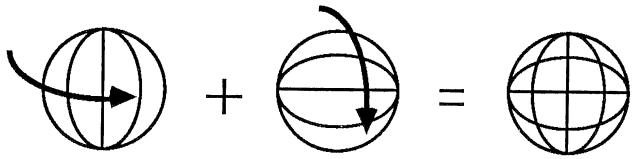
Before you build the ball, you must set up a coordinate system in *roll.c*. Make the system fairly large so the ball has a lot of room to roll. Edit *initialize*, and add this as the last line:

```
ortho(-400.0, 400.0, -400.0, 400.0, -400.0, 400.0);
```

Since the ball needs to roll, you should center it at the origin. Use the **circ** routine to make it round. The one problem is that **circ** comes only in the 2D variety. How do you make a 2D circle into a 3D sphere? Give it a few spins.

Modeling transformations are an important part of 3D modeling. You can create a sphere by rotating a circle, or a cone by scaling and translating it.

To create a sphere you use the `circ` drawing routine and the `rotate` modeling transformation. You need to rotate the circle about both the *x* and *y* axes to create a wireframe sphere that will look realistic when you "roll" it along the *z* axis.



You also need to use `pushmatrix` and `popmatrix` because you don't want the whole coordinate system to rotate when you draw your sphere. With these considerations in mind, create a *drawsphere* function at the end of your program.

```
drawsphere()
{
     int i;

/*-----------------------------------------------------
Save the current matrix for future use.
-----------------------------------------------------*/
     pushmatrix();

/*-----------------------------------------------------
Rotate the circle 15 degrees about the
y axis, then draw it.  Do this 24 times.
This creates the "lines of longitude".
-----------------------------------------------------*/
     for (i=0; i<=24; i++)
     {
          rotate(150, 'y');
          circ(0.0, 0.0, 20.0);
     }
```

```
/*-----------------------------------------------------
Push this matrix and undo the rotation.
-----------------------------------------------------*/
     pushmatrix();
     rotate(-150, 'y');

/*-----------------------------------------------------
Create the "lines of latitude".
-----------------------------------------------------*/
     for (i=0; i<=24; i++)
     {
          rotate(150, 'x');
          circ(0.0, 0.0, 20.0);
     }

/*-----------------------------------------------------
Delete the second rotation matrix, then
delete the first rotation matrix. Now
the original is current again.
-----------------------------------------------------*/
     popmatrix();
     popmatrix();
}
```

Making the Ball Move

First, you want the ball to start far away from you. Push it back along the negative *z* axis by using a **`translate`** in the *initialize* function. This will translate the ball only once when the program starts. Add this as the last line of *initialize*:

```
translate (0.0, 0.0, -300.0);
```

To make the ball seem to roll, you rotate it. To make it move towards you within its coordinate system, you translate it. And to make it look like it's coming towards you (to make it look bigger as it gets closer), you scale it. Add a function called *rollem* after *drawimage*.

```
rollem()
{
     rotate (150, 'x');
     scale(1.05, 1.05, 1.05);
     translate(0.0, 0.0, 1.0);

     drawsphere();
}
```

Finally, you need to call *rollem* from *drawimage* right before you call **swapbuffers**.

```
drawimage()
{
     color(BLACK);
     clear();

     color(RED);
     rollem();

     swapbuffers();
}
```

You can find this entire program in the file *rollem.c* which is in the *explore* directory. Compile and run your program.

Workshop

composite12.c uses the modeling transformations and matrix manipulation routines to make the ball move more realistically.

Summary

- When you work with 3D objects, you must define a 3D coordinate system in which they will exist.
- The IRIS stores transformation information in matrices. These matrices are stacked one on top of the other.
- The IRIS transformation matrix keeps track of modeling transformations. These transformations are cumulative, so the current matrix is constantly changing unless you use the matrix manipulation routines.
- `pushmatrix` copies the current matrix, pushes the original down one position on the stack, and puts the copy on top of the stack. `popmatrix` permanently deletes the current matrix and puts the matrix that was stored immediately below it on top of the stack.
- You can use modeling transformations not only to display models in different orientations, but also to build models (e.g., rotate a circle to make a sphere) and to animate them.

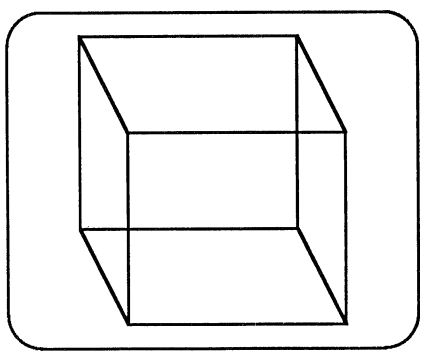
7 Changing the Point of View

The modeling transformations and matrices are fine for modifying and moving your models. However, sometimes it's not the model itself that you want to change — it's the angle or distance from which you are looking at the model. For this purpose the Graphics Library includes two types of routines: projection and viewing transformations. They let you change your point of view without changing your model.

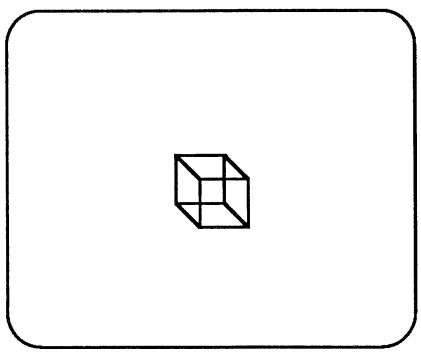
Projection Transformations

After you describe a 3D object to the IRIS, it figures out how to display it on your 2D screen. The 2D display is called a *projection* of the 3D object.

Projecting objects is similar to taking pictures. There is an original 3D object that you want to represent on a 2D surface. You focus on the object, move closer to or farther from it to adjust how much of the object's world (the background) will be in the picture, and then you capture the scene in 2D.



closeup



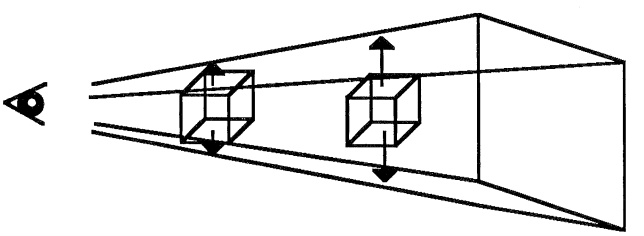
not so close up

The IRIS provides the same flexibility. All objects that you draw are already in focus, so your only concern is how much background to include. Once you decide, you can specify a *viewing volume*, which describes the boundaries of the object's world. A viewing volume can be any size, but can have only one of two shapes (see below). The shape you choose tells the IRIS how to transform the 3D object into a 2D projection.

Perspective and Orthographic Projections

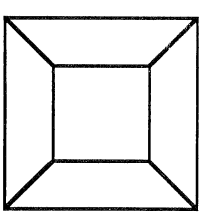
When you look out of your window at a car that is ten feet away, and at a house that is a mile away, the car looks as if it should be able to drive over the house without too much trouble. This is because you naturally view objects in perspective. An object that is close to you always looks larger than an identical object that is far away.

To capture this perspective realistically on the IRIS, use the `perspective` routine. It defines a pyramid-shaped viewing volume. Objects that are closer to you occupy a larger amount of the viewing volume than objects that are farther away. The closer objects are, the bigger they look.



perspective viewing volume

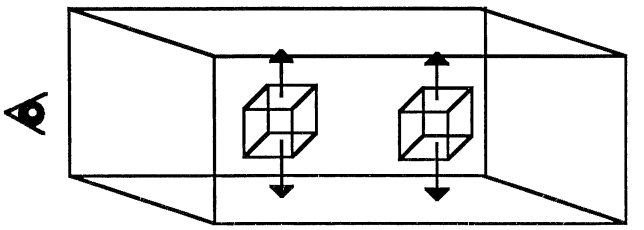
You can also see the effect of a perspective projection on an individual object. For example, the front face of a cube that is projected with perspective will look larger than its back face.



Recall the 3D cube you drew at the beginning of Chapter 6. It didn't look like this figure, it just looked like a square.

Part of the problem with your cube was that you were looking at it straight on — it wasn't rotated. The other part was that it was projected orthographically, rather than with perspective.

Orthographic projections make objects that are the same size look the same size, no matter how near or far away they are. When you use the `ortho` routine, you define a square or rectangular box as the viewing volume. The amount of viewing volume that any one object occupies depends only on its actual size.



orthographic viewing volume

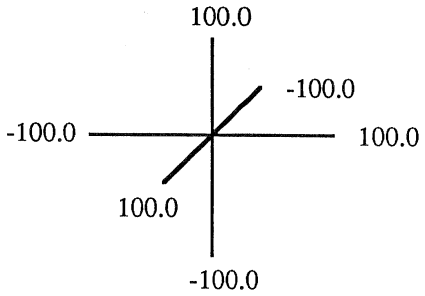
Using the ortho and perspective Routines

In the last chapter, you used `ortho` in the *initialize* function to set up a coordinate system. You were also defining a viewing volume that looks like the figure above. Return to your old *cube3d.c* program for a minute.

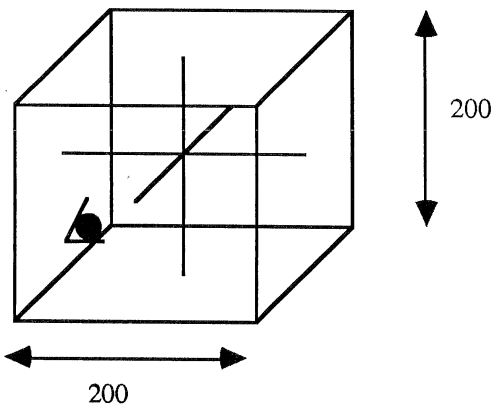
Your `ortho` routine should look like this:

```
ortho(-100.0, 100.0, -100.0, 100.0, -100.0, 100.0);
```

This defines a coordinate system that extends 100 units in both positive and negative directions, along all three axes.



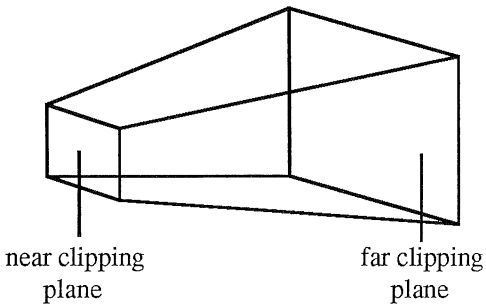
If you draw a cube around this system you see what the orthographic viewing volume looks like.



Notice that there are near and far boundaries (boundaries in the *z* direction) as well as left, right, bottom, and top boundaries. These are called *clipping planes*. *IRIS Graphics Programming* explains how the arguments to `ortho` establish a viewing volume. Rather than representing coordinates, the six arguments represent the clipping planes.

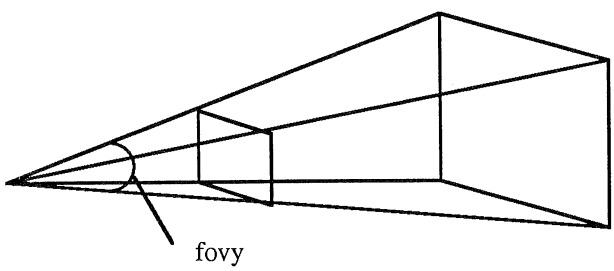
To project objects orthographically, you can use one other routine: `ortho2`. The only difference between `ortho` and `ortho2` is that `ortho2` doesn't take arguments for the near and far clipping planes. It assumes that their values are −1.0 and +1.0, respectively. `ortho2` is useful for 2D applications.

The `perspective` routine also has six clipping planes, so its viewing volume can look like a truncated pyramid.

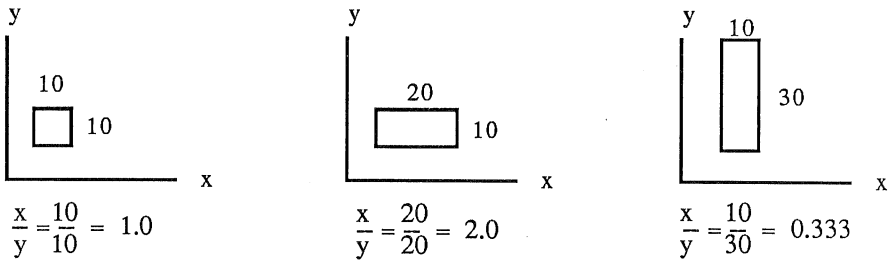


The arguments to `perspective` are different from those to `ortho`. `perspective` needs four pieces of information:

- The field of view (*fovy*). This is the angle that would form the apex of the viewing volume were it not truncated. *fovy* affects the location of the right, left, bottom, and top clipping planes, and the amount of perspective distortion. Typically, this angle is between 40 and 130 degrees, although legal angles are 0 to 180 degrees.

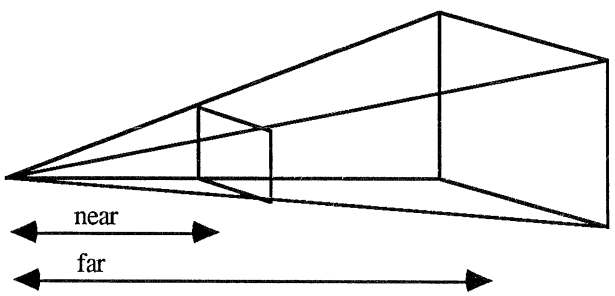


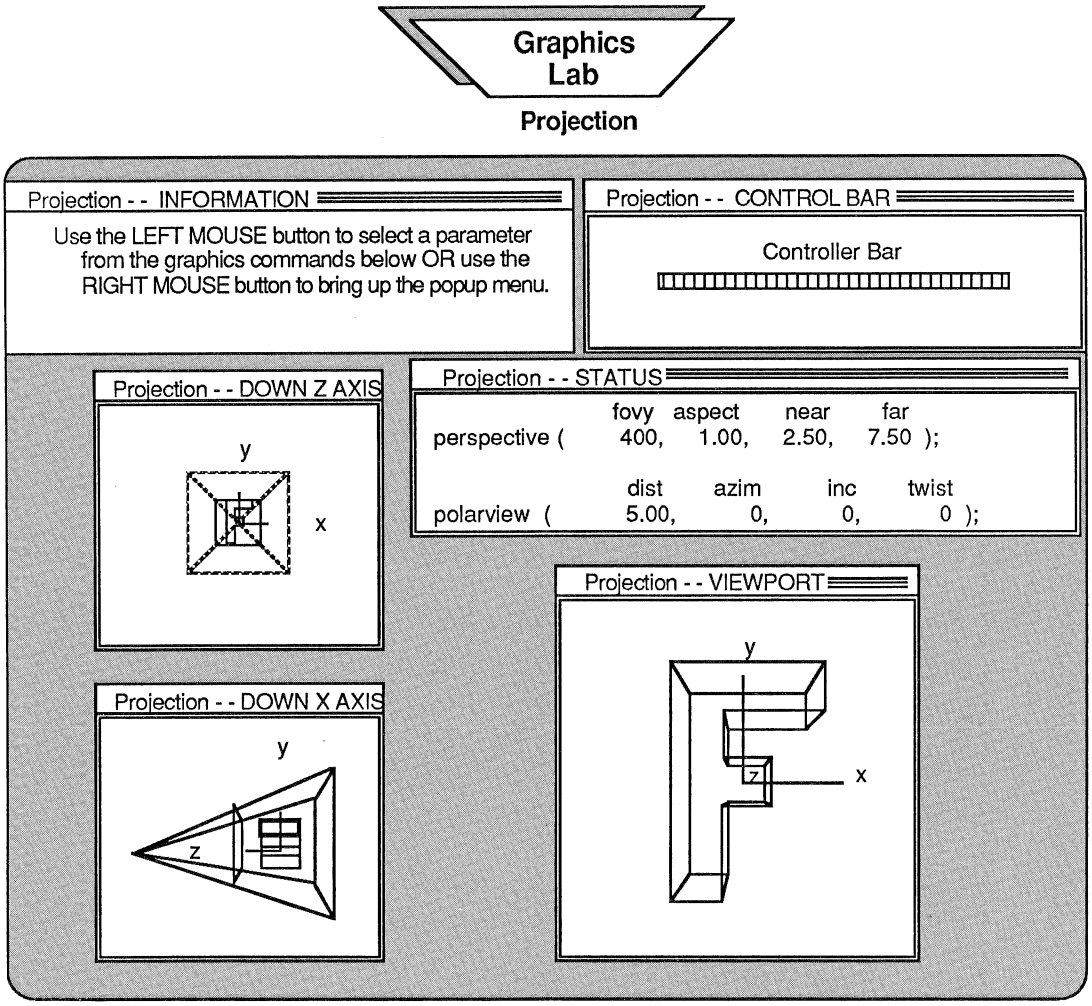
- The aspect ratio (*aspect*). This is the ratio of *x* to *y*. It affects the location of the left, right, bottom, and top clipping planes. Usually it's 1:1 (1.0), but you can use it to distort your images to create interesting visual effects.



$\frac{x}{y} = \frac{10}{10} = 1.0$ $\frac{x}{y} = \frac{20}{20} = 2.0$ $\frac{x}{y} = \frac{10}{30} = 0.333$

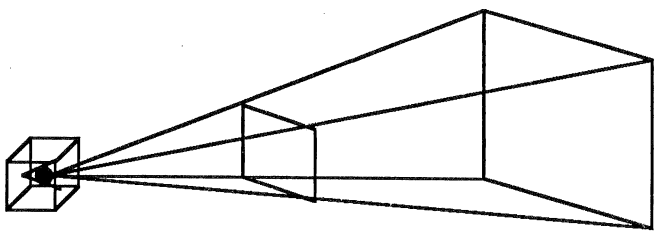
- The *near* and *far* coordinates. These are the distances from the apex of the pyramid to the near and far clipping planes. Both values are usually positive.





Run *projection*. Choose "perspective" from the pop-up menu, then change the parameters for it. Notice how the clipping planes work. Change the parameters of `perspective` to see how they affect the image. Also choose "ortho" to see how it affects the image. (Your IRIS must have at least 12 bitplanes to run this Lab.)

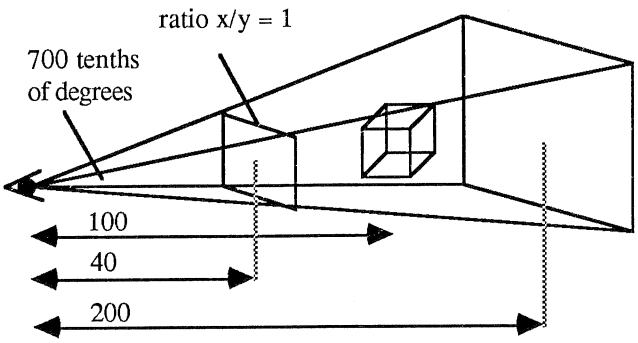
The last difference between **perspective** and **ortho** is that **perspective** assumes that the apex of its pyramid is positioned at (0, 0, 0), and that the viewer is looking down the negative *z* axis. This means that an object centered at the origin (like your cube) may not be displayed if you substitute the **perspective** routine for the **ortho** routine. Your eye would be positioned at the exact center of the cube, and your field of view wouldn't be wide enough for any of the sides to be displayed.



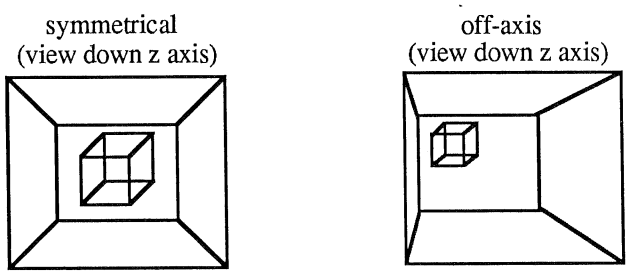
There are some special routines for solving this problem that you'll learn about in the next section. For now, you can translate the cube down the *z* axis to put it within the viewing volume. Make a copy of *cube3d.c* and call it *percube3d.c*. Edit this new file so the cube is projected with perspective. Delete the line that contains the **ortho** routine, and replace it with these lines.

```
/*-----------------------------------------------------
  fovy = 70 degrees; aspect ratio is 1:1; near plane
  is 40 units from the viewer; far plane is 200
  units from the viewer.
-----------------------------------------------------*/
perspective(700, 1.0, 40.0, 200.0);

/*-----------------------------------------------------
  Push cube 100 units down -z axis.
-----------------------------------------------------*/
translate(0.0, 0.0, -100.0);
```

Now your cube is positioned like this:



To project objects with perspective, you can use one other routine: `window`. It takes the same arguments as `ortho` (*left, right, bottom, top, near, far*). Like `perspective`, the apex of its pyramid is positioned at (0, 0, 0). The difference is that `window` can specify a viewing volume that is not symmetrical. This creates an interesting visual effect called an *off-axis* view.



If you use `window` to define a symmetrical viewing volume, it looks identical to a volume defined with `perspective`.

Run the Graphics Lab *projection* again, but this time choose "window". Change its parameters to create an off-axis view.

Gaining More Perspective

Using **perspective** not only makes the objects on the screen look more realistic, it also makes the way you visualize your program more natural. Take your *rollem* program from Chapter 6. You rotated, translated, and scaled the ball to make it look like it was coming towards you. Why did you have to scale the ball? To create the illusion of perspective.

Now that you can define a viewing volume that shows perspective, you don't have to increase the size of the ball manually as it comes towards you. Make a copy of *roll.c* or *rollem.c* and call it *rollp.c*. Edit *rollp.c* so it uses **perspective**, and no longer uses **scale**. This is a two-step process.

1. Tell the IRIS that the object's world has perspective, then call **translate** so the objects will appear within the viewing volume. Delete the line in *initialize* that contains the **ortho** routine. Replace it with these lines:

```
perspective(900, 1.0, 50.0, 500.0);
translate(0.0, 0.0, -400.0);
```

2. Now you can eliminate the **scale** routine, and let **translate** and **perspective** take care of making the ball look like it's coming towards you. Edit the *rollem* function so it looks like this:

```
rollem()
{
     rotate(150, 'x');
     translate(0.0, 0.0, 1.0);

     drawsphere();
}
```

This didn't work very well. The ball is rotating, but it looks more like it's orbiting than rolling. Note that in the original *rollem.c*, **translate** didn't actually move the ball towards you — it was just an illusion created by **scale**. When the *rollem* function rotates the ball, this rotation is added to the matrix. Now the ball's coordinate system is rotated, so a **translate** down the *z* axis will move the ball in the new direction.

To prevent this from happening, use the matrix manipulation routines to control the rotations. Remember, when you pop a matrix, all previous transformations are deleted. This means if you pop a matrix that contains the last **translate**, the ball will go back to its original position (0.0, 0.0, -400.0). You can solve this problem by declaring a static integer to keep track of how far the ball has been translated. Edit the function *rollem* so it looks like this:

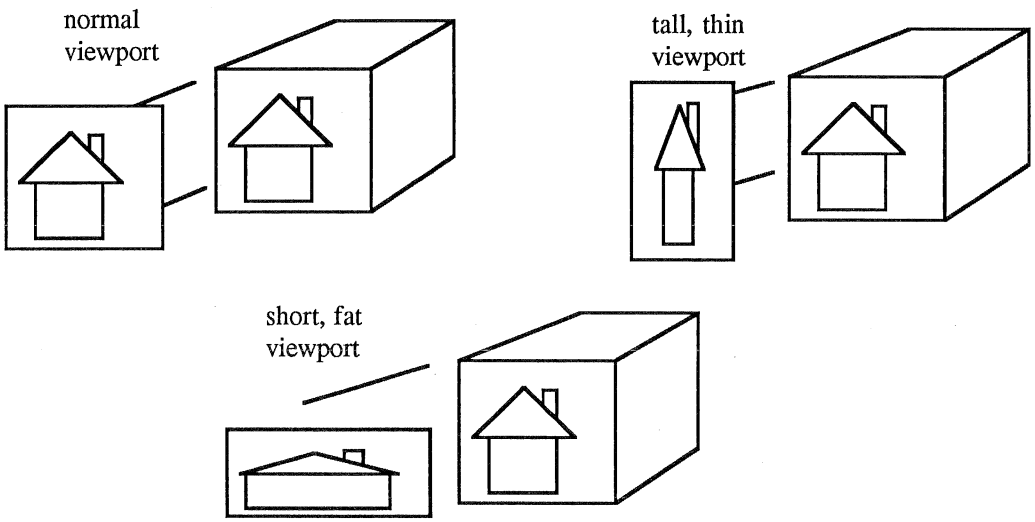
```
rollem()
{
     static int i = 0;
     translate(0.0, 0.0, 1.5);
     pushmatrix();
          rotate(75*i++, 'x');
          drawsphere();
     popmatrix();
}
```

Compile and run your program. How does it compare to the orthographic version?

Viewports

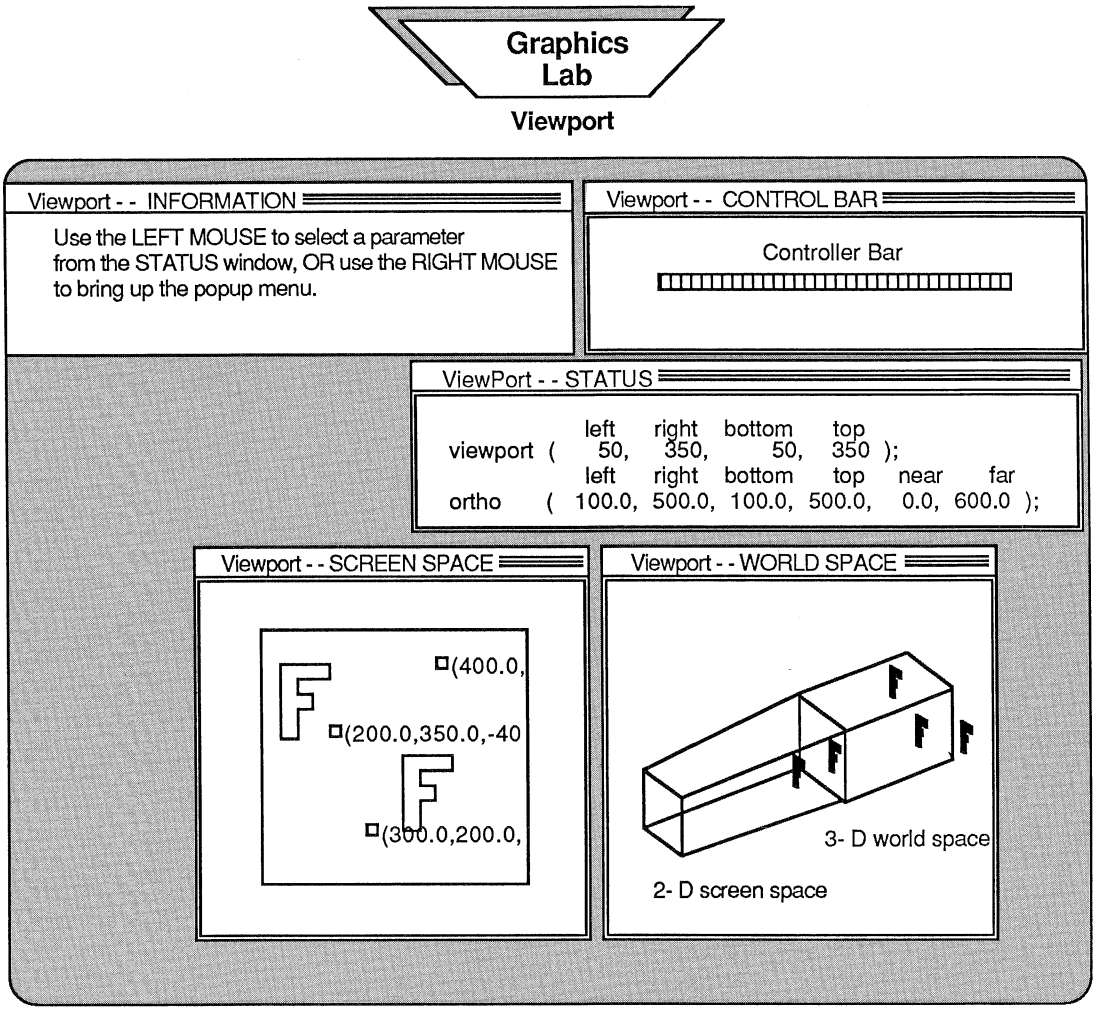
Just as you can establish boundaries for a 3D viewing volume, you can specify the size and shape of the 2D area that displays it. This area is called the *viewport.*

The shape you specify for a viewport should be the same as the shape of the near clipping plane. Otherwise, your image will be distorted.



There are two default viewport settings on the IRIS. If your program doesn't run under *mex*, the default viewport is the entire screen (`viewport (0, XMAXSCREEN, 0, YMAXSCREEN);`). If it does run under *mex*, the lower left corner of the window is considered (0, 0), and the upper right corner (*x, y*) is determined by how large an area you sweep out (`viewport (0, x, 0, y)`).

Another useful routine for preventing distortion is `keepaspect`. It lets you specify an aspect ratio for the shape of the viewport. You put it in the *initialize* function, before you call `winopen`. It was introduced in the Workshop *coord10.c*.



Run *viewport*. It shows how `ortho` and `viewport` work together. (Your IRIS must have at least 12 bitplanes to run this Lab.)

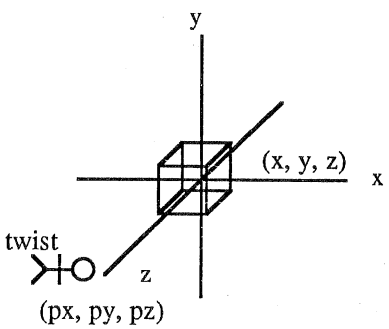
Viewing Transformations

Up to this point, the viewer has always been looking in the same direction — straight down the negative *z* axis. Sometimes you'll want to change the point of view, that is, place the viewer somewhere else within the viewing volume. To do this, you use viewing transformations.

Think of the viewing volume as the *world space* for your objects, and your viewer as a person who lives in the same world. You can move the person around the world in different ways using two Graphics Library routines: `lookat` and `polarview`.

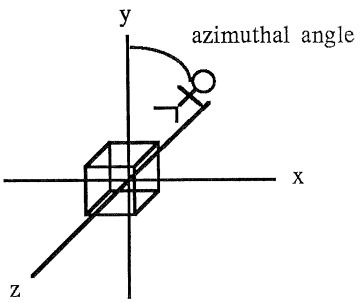
Two New Angles

`lookat` lets you establish a line of sight with a counterclockwise twist. You specify the location of the person (*x*, *y*, and *z* coordinates), the location of a point that the person is viewing (*x*, *y*, and *z* coordinates) plus a clockwise angle of twist (tenths of degrees). This twist is the amount of rotation about the line of sight, which is the straight line between the viewer and the object.

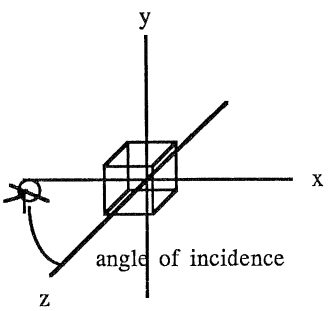


`polarview` assumes that the object being viewed is located at the origin, and the person can be anywhere in the world space. You specify the person's position with four arguments: the distance between the person and the object (a floating point number), the angle of incidence, the azimuthal angle, and an angle of twist.

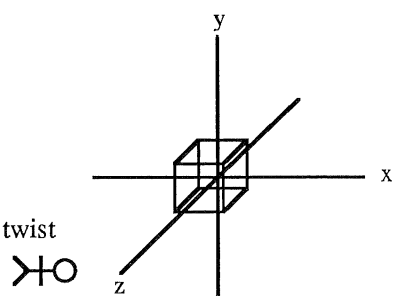
The azimuthal angle lies in the *x,y* plane. It is the difference between the object's orientation on the *y* axis (0 degrees) and the orientation of the viewer in the *x,y* plane. The direction of rotation follows the left-hand rule.



The angle of incidence lies in the *y,z* plane. It is the difference between the object's orientation on the *z* axis (0 degrees) and the orientation of the viewer in the *y,z* plane. Like the azimuthal angle, the rotation follows the left-hand rule.



The angle of twist is an amount of clockwise rotation about the line of sight (the straight line between the viewer and the object).



Run the Graphics Lab *projection* again. Try out the viewing transformations.

Workshop

view13.c lets you see the diamond and the ball from two different angles. First you see the diamond and the hit from your usual angle. Then, after a short pause, you see an instant replay from a new vantage point established with `polarview`.

Using lookat and polarview

Now you can use these routines to look at your rolling ball from a new angle. Start with **lookat**.

Edit your *rollp.c* program so your viewer is standing halfway between where the ball starts rolling and where it stops. Delete the line in *initialize* that contains **translate**. After the line that contains **perspective**, add this line:

```
lookat(3.0 ,0.0 ,9.0, 0.0, 0.0, 0.0);
```

To place yourself behind the ball and watch it roll away, replace the **lookat** line above with this line:

```
lookat(0.0, 0.0, -450.0, 0.0, 0.0, 0.0);
```

Now try **polarview**. Make the ball roll towards you, up a board that is inclined 30 degrees. Replace the **lookat** line with this:

```
polarview(0, 0, 300, 0);
```

Experiment with these routines. Try them together and see how they affect each other.

Workshop

parabola14 adds one more piece of realism to the baseball program. The ball travels in a parabolic path from the batter's box to the ground.

More on Matrices

In Chapter 6 you learned how to manipulate the matrices that keep track of modeling transformations. Both projection and viewing transformations use the same matrices, and you can manipulate them using the same routines as before.

Projection transformations affect the current matrix differently than do modeling transformations. When you call a projection transformation, the current matrix is destroyed and is replaced by a new matrix that contains only the projection transformation. Nothing happens to any other matrices on the stack. So, if you want to save your current matrix, call a `pushmatrix` before you call a projection transformation.

Viewing transformations affect the current matrix the same way as modeling transformations — they change it cumulatively. When you call a viewing transformation, you change the current matrix, but you don't destroy it.

Because the transformations affect the matrix differently, the order in which you call them is very important.

1. Always call the projection transformation first, because it completely destroys the current matrix.
2. Call the viewing transformation. You must establish the orientation of the object's world before you can move the object around.
3. Call the modeling transformations.

If you look at your code for *rollp.c*, you see that the transformations are called in this order. Change the order around and see what happens.

Summary

- Projection transformations determine how the 3D model is displayed on your 2D screen.
- You can project models either orthographically or with perspective.
- Viewing transformations let you change the viewer's position within the viewing volume; that is, the position relative to the 3D world.
- All transformations (projection, viewing, and modeling) use the same matrices. Both modeling and viewing transformations are cumulative and change only the current matrix; projection transformations destroy the current matrix and replace it with a new one.
- You must call the transformations in this order:
    1. projection
    2. viewing
    3. modeling

8 Where to Go from Here

This tutorial provides a basic introduction to the IRIS Graphics Library and to real-time 3D programming on the IRIS. You're now prepared to go on to more advanced topics in IRIS graphics programming. This chapter tells you where to find more information.

The IRIS Graphics Library

For more information on the IRIS Graphics Library, see *IRIS User's Guide*, Volumes I and II. Volume I contains two major parts:

- *IRIS Graphics Programming* is a narrative description of the Graphics Library routines, arranged by subject matter. Use this part of the manual to find the Graphics Library routine that matches your programming task.
- *Using mex, the IRIS Window Manager* describes the window manager routines contained in the Graphics Library.

Volume II of *IRIS User's Guide* contains *IRIS Reference Manual*, which is arranged alphabetically by routine. Once you know the name of a Graphics Library routine, look here for its syntax and a brief description.

General Graphics References

For general information on computer graphics, consult these textbooks:

- *Procedural Elements for Computer Graphics*, David F. Rogers, McGraw-Hill Book Company, 1985. Introductory textbook.
- *Fundamentals of Interactive Computer Graphics* (Addison-Wesley Systems Programming Series), James D. Foley and Andries Van Dam, Addison-Wesley Publishing Company, 1982. General textbook.
- *Principles of Interactive Computer Graphics*, William M. Newman and Robert F. Sproull, McGraw-Hill Book Company, 1979. General textbook.
- *Mathematical Elements for Computer Graphics*, David F. Rogers and J. Alan Adams, McGraw-Hill Book Company, 1976. Textbook for transformations and curves.

Advanced Graphics Labs

You've been following the on-line Graphics Labs as you worked through this tutorial. Now you're ready for Graphics Labs on more advanced topics:

- *backface* shows two views of a cube, contrasting images with and backface removal, a form of hidden surface removal. (12 bitplanes required)
- *curve* shows how to use the curve routines in the Graphics Library. (12 bitplanes required)
- *depthcue* shows a depthcued, wireframe model of the letter 'F'. This lab demonstrates the effects and interactions of the IRIS Graphics Library routines `perspective`, `setdepth`, and `shaderange`. (16 bitplanes required)
- *gamma* shows the effects of gamma correcting a color ramp. You control slider bars to change interactively the value of the gamma constant. (16 bitplanes required)
- *gouraud* illustrates the concept of Gouraud shading by simulating a model polygon with very large pixels. You can edit the position and intensity of each vertex of a four-sided polygon. (16 bitplanes required)
- *patch* shows how to use the patch routines in the Graphics Library. Do the *curve* lab before this one; the two labs have similar structures. (12 bitplanes required)
- *scrmask* shows the effects of the IRIS Graphics Library routines `viewport` and `scrmask` on text strings. (12 bitplanes required)
- *zbuffer* shows the effects of the IRIS Graphics Library `zbuffer` routine. This lab shows the difference between a solid object that uses the z buffer for hidden surface removal, and a solid object that has no hidden surface removal. (32 bitplanes required)

SiliconGraphics
Computer Systems

2011 Stierlin Road
Mountain View, California 94043
Telephone (415) 960-1980

P/N 007-1103-010 Printed in U.S.A. 7/86