IRIS User's Guide
Volume I
Programming Guide

Version 4.0

Document Number 007-1101-040

© Copyright 1987, Silicon Graphics, Inc. - All rights reserved

This document contains proprietary and confidential information of Silicon Graphics, Inc., and is protected by Federal copyright law. The contents of this document may not be disclosed to third parties, copied or duplicated in any form, in whole or in part, without the express written permission of Silicon Graphics, Inc.

U.S. Government Limited Rights

Use, duplication or disclosure of the technical data contained in this document by the Government is subject to restrictions as set forth in subdivision (b) (2) (ii) of the Rights in Technical Data and Computer Software clause at 52.227-7013. Contractor/manufacturer is Silicon Graphics Inc., 2011 Stierlin Road, Mountain View, CA 94039-7311.

IRIS User's Guide
Volume I
Programming Guide
Version 4.0
Document Number 007-1101-040

Silicon Graphics, Inc.
Mountain View, California

UNIX is a trademark of AT&T Bell Laboratories.

Contents

VOLUME I

GRAPHICS PROGRAMMING

1. Introduction 1-1
- 1.1 System Overview 1-1
- 1.2 The Graphics Library 1-3
- 1.3 Documentation Conventions 1-5
- 1.4 Related Publications 1-6

2. Global State Attributes 2-1
- 2.1 Initialization 2-1
- 2.2 Saving Global State Attributes 2-7

3. Drawing Routines 3-1
- 3.1 Current Drawing Positions 3-2
- 3.2 Clearing the Viewport 3-3
- 3.3 Points 3-3
- 3.4 Lines 3-5
  - 3.4.1 Relative Drawing 3-7
- 3.5 Rectangles 3-8
- 3.6 Polygons 3-10
- 3.7 Circles and Arcs 3-13
  - 3.7.1 Circles 3-13
  - 3.7.2 Arcs 3-14
- 3.8 Text 3-16
- 3.9 Writing and Reading Pixels 3-19

4. Coordinate Transformations 4-1
- 4.1 Modeling Transformations 4-2
- 4.2 Viewing Transformations 4-7
- 4.3 Projection Transformations 4-10
- 4.4 Viewports 4-15
- 4.5 User-Defined Transformations 4-20

5. **Linestyles, Patterns, and Fonts** 5-1
5.1 Linestyles 5-1
5.1.1 Modifying the Linestyle Pattern 5-3
5.2 Patterns 5-6
5.3 Fonts 5-9

6. **Display and Color Modes** 6-1
6.1 Display Modes 6-1
6.1.1 RGB Mode 6-3
6.1.2 Single Buffer and Double Buffer Modes 6-3
6.2 Color Maps 6-8
6.3 Colors and Writemasks 6-12
6.3.1 Colors 6-12
6.3.2 Writemasks 6-16
6.4 Cursors 6-19

7. **Input/Output Routines** 7-1
7.1 Polling and Queueing 7-1
7.2 Initializing a Device 7-4
7.3 Polling a Device 7-6
7.4 The Event Queue 7-7
7.5 Controlling Peripheral Input/Output Devices 7-11
7.6 Special Devices 7-14
7.6.1 Keyboard Devices 7-14
7.6.2 Timer Devices 7-14
7.6.3 Cursor Devices 7-15
7.6.4 Ghost Devices 7-15
7.6.5 Window Manager Devices 7-15

8. **Graphical Objects** 8-1
8.1 Defining An Object 8-1
8.2 Using Objects 8-5
8.3 Object Editing 8-8
8.3.1 Identifying Display List Items with Tags 8-10

8.3.2 Inserting, Deleting, and Replacing within Objects 8-13
8.3.3 Example 8-15
8.3.4 Object Memory Management 8-16

9. Picking and Selecting 9-1
9.1 Mapping Screen Coordinates to World Coordinates 9-1
9.2 Picking 9-2
9.2.1 Using the Name Stack 9-6
9.2.2 Defining the Picking Region 9-9
9.2.3 Example 9-10
9.3 Selecting 9-14

10. Geometry Pipeline Feedback 10-1
10.1 The Geometry Pipeline 10-1
10.2 Feedback Mode 10-6

11. Curves and Surfaces 11-1
11.1 Curve Mathematics 11-2
11.1.1 Bezier Cubic Curve 11-3
11.1.2 Cardinal Spline Cubic Curve 11-4
11.1.3 B-Spline Cubic Curve 11-6
11.2 Drawing Curves 11-7
11.2.1 Rational Curves 11-21
11.3 Drawing Surfaces 11-23

12. Hidden Surfaces 12-1
12.1 Z-Buffer Mode 12-1
12.2 Backfacing Polygon Removal 12-6

13. Shading and Depth-Cueing 13-1
13.1 Shading 13-1
13.2 Depth-Cueing 13-6

14. Textports 14-1

USING MEX, THE IRIS WINDOW MANAGER

1. **Getting Started with *mex*** W-1
   - 1.1 What Is a Window Manager? W-1
   - 1.2 Window Manager Terminology W-2
   - 1.3 What Does *mex* Do? W-3
   - 1.4 Creating Windows W-3
   - 1.5 Interacting with the Window Manager W-4
     - 1.5.1 Attaching to Windows W-5
     - 1.5.2 Selecting a Window W-5
     - 1.5.3 Moving a Window W-5
     - 1.5.4 Reshaping a Window W-6
     - 1.5.5 Pushing and Popping Windows W-6
     - 1.5.6 Removing a Window W-6
   - 1.6 Selecting a Title Font W-7
   - 1.7 Attaching to the Window Manager W-7
   - 1.8 Executing Graphics Programs from *mex* W-8

2. **Programming with *mex*** W-9
   - 2.1 Opening and Closing Windows W-9
   - 2.2 Setting Window Constraints W-11
     - 2.2.1 Setting Constraints for Existing Windows W-16
   - 2.3 Changing Windows Noninteractively (from within a Program) W-17
   - 2.4 Other Window Routines W-19
   - 2.5 Programming Hints W-24
     - 2.5.1 Graphics Initialization W-24
     - 2.5.2 Shared Facilities W-24
     - 2.5.3 Raster Fonts W-24
     - 2.5.4 The Event Queue W-25
   - 2.6 Sample Program: Single Buffer Mode W-25
   - 2.7 Sample Program: Double Buffer Mode W-27

3. **Making Pop-up Menus** W-31
   - 3.1 Creating a Pop-up Menu W-32
   - 3.2 Calling Up a Pop-up Menu W-34
   - 3.3 Choosing Colors for Pop-up Menus W-36
   - 3.4 Advanced Menu Formats W-38

3.4.1 Getting Back the Default Values for Menu Selections W-38
3.4.2 Changing the Return Values for Menu Selections W-39
3.4.3 Making a Title W-39
3.4.4 Binding a Function to a Whole Menu W-39
3.4.5 Binding a Function to a Menu Entry W-40
3.4.6 Making a Nested (Rollover) Menu W-40
3.5 An Example from *cedit*, a Color Editing Program W-41
3.6 Sample Program W-43

4. Customizing *mex* W-47
4.1 Interpreting Button Events W-48
4.2 Binding Colors to the Title Bar and Borders W-51
4.3 Binding Colors to Pop-up Menus and the Cursor W-53
4.4 Setting Color Map Entries W-53
4.5 A Sample *.mexrc* W-53
4.6 Arranging Your Desktop W-54

5. Controlling Multiple Windows from a Single Process W-57

SAMPLE CODE - EXAMPLES

1: Getting Started S-1

2: Common Drawing Commands S-5

3: Object Coordinate Systems S-15

4: Double Buffer Display Mode S-19

5: Input/Output Devices S-23

6: Interactive Drawing S-29

7: Pop-up Menus S-33

8: Modeling Transformations S-47

9: Writemasks and Color Maps S-61

10: A Color Editor S-67

SAMPLE CODE - WORKSHOPS

1: diamond1.c S-75

2: color2.c S-79

3: double3.c S-83

4: overlay4.c S-87

5: poll5.c S-91

6: aim6.c S-95

7: queue7.c S-99

8: menu8.c S-105

9: threed9.c S-111

10: coord10.c S-117

11: translate11.c S-123

12: composite12.c S-129

13: view13.c S-135

14: parabola14.c S-141

GLOSSARY GL-1

INDEX I-1

VOLUME II

REFERENCE MANUAL
Introduction
Permuted Index
Manual Pages

APPENDICES

A: Type Definitions for C and FORTRAN A-1
A.1 C Definitions A-1
A.2 FORTRAN Definitions A-19

B: Geometry Engine Computations B-1

C: Transformation Matrices C-1
C.1 Translation C-1
C.2 Scaling and Mirroring C-1
C.3 Rotation C-2
C.4 Viewing Transformations C-2
C.5 Perspective Transformations C-3
C.6 Orthographic Transformations C-4

D: Feedback Parser D-1

E: Window Manager Programs E-1
E.1 Color Tools E-1
E.2 Image Tools E-4
E.3 General Desktop Tools E-5
E.4 Utilities E-6
E.5 Device Usage Examples E-7
E.6 Fantasy Demonstrations E-8

F: IRIS Programming Tutorial Manual Pages F-1

G: Fast Immediate Mode and User-Defined Display Lists G-1
G.1 Introduction and Overview G-1
G.2 User-Defined Display Lists G-7
G.3 Example -- Defining Your Own Display List G-10
G.4 Counting Instructions G-15
G.5 Conclusions G-16

H: Using the Image Library H-1
H.1 Image Files H-1
H.2 Opening and Closing an Image File H-1
H.2.1 iopen - Open an Image File H-2
H.2.2 iclose - Close an Image File H-3
H.3 Reading from and Writing to Image Files H-3
H.3.1 putrow - Write a Row of Pixels from Buffer to Image File H-3
H.3.2 getrow - Read a Row of Pixels from Image File to Buffer H-4
H.4 Miscellaneous Functions H-4
H.4.1 isetname - Name an Image File H-4
H.4.2 isetcolormap - Interpret Pixel Values H-5
H.4.3 scrsave - Save Rectangular Region of Screen to Image File H-5
H.5 An Example H-6

GRAPHICS PROGRAMMING

GRAPHICS PROGRAMMING

1. **Introduction** 1-1
   - 1.1 System Overview 1-1
   - 1.2 The Graphics Library 1-3
   - 1.3 Documentation Conventions 1-5
   - 1.4 Related Publications 1-6

2. **Global State Attributes** 2-1
   - 2.1 Initialization 2-1
   - 2.2 Saving Global State Attributes 2-7

3. **Drawing Routines** 3-1
   - 3.1 Current Drawing Positions 3-2
   - 3.2 Clearing the Viewport 3-3
   - 3.3 Points 3-3
   - 3.4 Lines 3-5
     - 3.4.1 Relative Drawing 3-7
   - 3.5 Rectangles 3-8
   - 3.6 Polygons 3-10
   - 3.7 Circles and Arcs 3-13
     - 3.7.1 Circles 3-13
     - 3.7.2 Arcs 3-14
   - 3.8 Text 3-16
   - 3.9 Writing and Reading Pixels 3-19

4. Coordinate Transformations 4-1
4.1 Modeling Transformations 4-2
4.2 Viewing Transformations 4-7
4.3 Projection Transformations 4-10
4.4 Viewports 4-15
4.5 User-Defined Transformations 4-20

5. Linestyles, Patterns, and Fonts 5-1
5.1 Linestyles 5-1
5.1.1 Modifying the Linestyle Pattern 5-3
5.2 Patterns 5-6
5.3 Fonts 5-9

6. Display and Color Modes 6-1
6.1 Display Modes 6-1
6.1.1 RGB Mode 6-3
6.1.2 Single Buffer and Double Buffer Modes 6-3
6.2 Color Maps 6-8
6.3 Colors and Writemasks 6-12
6.3.1 Colors 6-12
6.3.2 Writemasks 6-16
6.4 Cursors 6-19

7. Input/Output Routines 7-1
7.1 Polling and Queueing 7-1
7.2 Initializing a Device 7-4
7.3 Polling a Device 7-6
7.4 The Event Queue 7-7
7.5 Controlling Peripheral Input/Output Devices 7-11
7.6 Special Devices 7-14
7.6.1 Keyboard Devices 7-14
7.6.2 Timer Devices 7-14
7.6.3 Cursor Devices 7-15
7.6.4 Ghost Devices 7-15
7.6.5 Window Manager Devices 7-15

8. Graphical Objects 8-1
8.1 Defining An Object 8-1
8.2 Using Objects 8-5
8.3 Object Editing 8-8
8.3.1 Identifying Display List Items with Tags 8-10
8.3.2 Inserting, Deleting, and Replacing within Objects 8-13
8.3.3 Example 8-15
8.3.4 Object Memory Management 8-16

9. Picking and Selecting 9-1
9.1 Mapping Screen Coordinates to World Coordinates 9-1
9.2 Picking 9-2
9.2.1 Using the Name Stack 9-6
9.2.2 Defining the Picking Region 9-9
9.2.3 Example 9-10
9.3 Selecting 9-14

10. Geometry Pipeline Feedback 10-1
10.1 The Geometry Pipeline 10-1
10.2 Feedback Mode 10-6

11. Curves and Surfaces 11-1
11.1 Curve Mathematics 11-2
11.1.1 Bezier Cubic Curve 11-3
11.1.2 Cardinal Spline Cubic Curve 11-4
11.1.3 B-Spline Cubic Curve 11-6
11.2 Drawing Curves 11-7
11.2.1 Rational Curves 11-21
11.3 Drawing Surfaces 11-23

12. Hidden Surfaces 12-1
12.1 Z-Buffer Mode 12-1
12.2 Backfacing Polygon Removal 12-6

13. Shading and Depth-Cueing 13-1
13.1 Shading 13-1
13.2 Depth-Cueing 13-6

14. Textports 14-1

1. Introduction

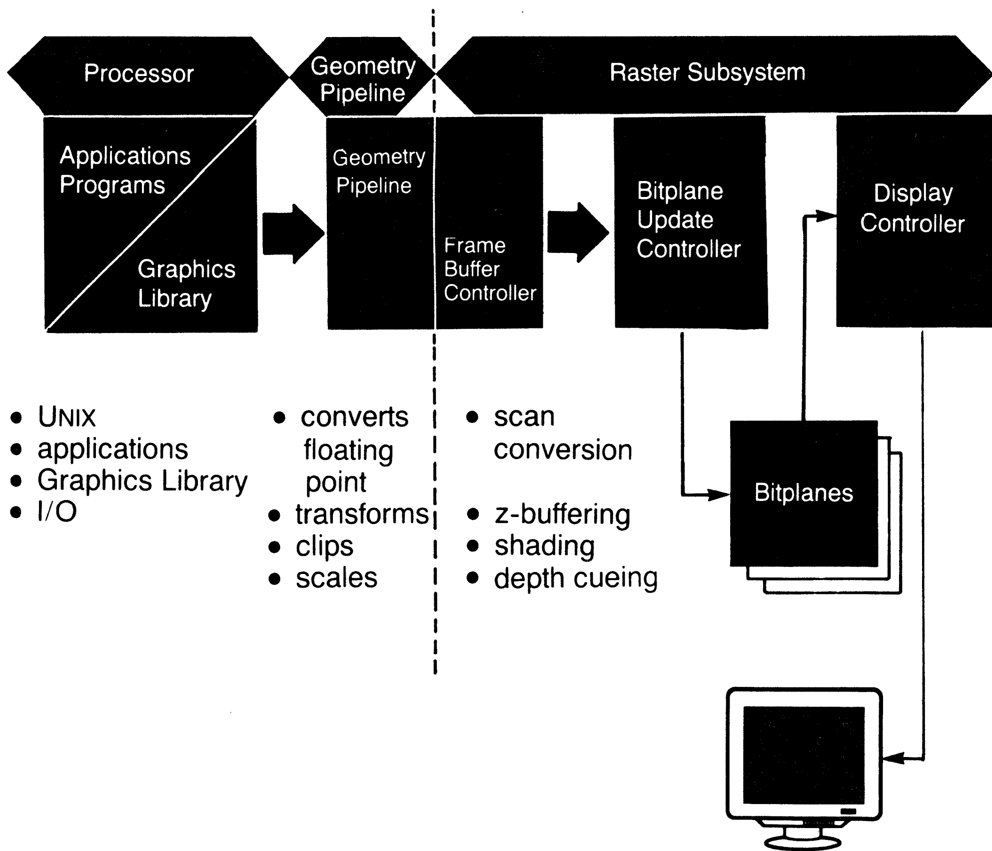
The IRIS (Integrated Raster Imaging System) workstation is a high-performance, high-resolution color computing system for 2-D and 3-D computer graphics. It provides a powerful set of graphics primitives in a combination of custom VLSI circuits, conventional hardware, firmware, and software.

The heart of the system is a custom VLSI chip called the *Geometry Engine*™. A pipeline of 10 or 12 Geometry Engines accepts points, vectors, polygons, characters, and curves in user-defined coordinate systems and transforms them to screen coordinates, with the use of rotation, clipping, and scaling. In addition to the Geometry Pipeline, the system consists of a general purpose microprocessor, a raster subsystem, a high-resolution color monitor, a keyboard, and graphics input devices.

1.1 System Overview

Conceptually, the graphics hardware of the system is divided into three pipelined components shown in Figure 1-1: the applications/graphics processor, the Geometry Pipeline, and the raster subsystem. The applications/graphics processor runs the applications program, and controls the Geometry Pipeline and the raster subsystem. Graphics routines issued by the applications program either are sent immediately through the pipeline, or are compiled into graphical objects (display lists of graphics routines) which you can call later.

Graphics routines are expressed in 2-D or 3-D user-defined coordinates. These routines go through the Geometry Pipeline, which performs matrix transformations on the coordinates, clips the coordinates to normalized coordinates, and scales the transformed, clipped coordinates to screen or window coordinates. The output of the Geometry Pipeline is then sent to



The IRIS graphics hardware consists of three subsystems: the applications/graphics processor, the Geometry Pipeline, and the raster subsystem.

Figure 1-1. IRIS Graphics Hardware

the raster subsystem. The raster subsystem fills in the pixels between the endpoints of the lines and the interiors of polygons; draws bit-mapped characters, and performs shading, depth-cueing, and hidden surface removal. A color value for each pixel is stored in the bitplanes. The system uses the values contained in the bitplanes to display an image on the monitor.

1.2 The Graphics Library

The Graphics Library is a set of graphics and utility routines that provide high- and low-level support for graphics. The system software is written in C, although you can call the routines in C, FORTRAN, and Pascal. The graphics routines are grouped into the following categories by chapter:

- *Global state* routines, which initialize the hardware and control global state attributes (Chapter 2)
- *Drawing* routines, which draw points, lines, polygons, circles, arcs, and text strings, and select characteristics for drawing lines, filling polygons, and writing text strings (Chapter 3)
- *Coordinate transformation* routines, which perform manipulations on coordinate systems, including the mapping of user-defined coordinate systems to screen coordinate systems (Chapter 4)
- *Pattern and font* routines, which define linestyle, patterns, and fonts (Chapter 5)
- *Display mode and color* routines, which determine how the bitplane image memory is used and how objects are colored (Chapter 6)
- *Input/output* routines, which initialize and read input/output devices (Chapter 7)
- *Object creation and editing* routines, which provide the means to create graphical objects and hierarchical structures of graphics routines (Chapter 8)
- *Picking and selecting* routines, which identify the routines that draw to a specified area of the screen (Chapter 9)
- *Geometry Pipeline feedback* commands, which provide access to the computing capabilities of the geometry hardware (Chapter 10)

- *Curve and surface* routines, which draw curved lines and wire-frame surfaces (Chapter 11)
- *Hidden surface* routines, including z-buffer mode, in which hidden lines and surfaces are removed from an image, and backface mode, in which backfacing polygons are removed from an image (Chapter 12)
- *Shading* routines, which draw Gouraud-shaded polygons, and *depth-cueing* routines, which draw points, lines, curves, and surfaces with intensities that vary as a function of depth to enhance 3-D display (Chapter 13)
- *Textport* routines, which allocate an area of the screen for writing text (Chapter 14)
- *Window manager* routines, which create and manipulate graphics windows (see *Using mex*)

Additional material is covered in the appendices:

- Appendix A contains header files for graphics programs.
- Appendix B shows the computations performed by the Geometry Engines to transform, clip, and scale coordinate data.
- Appendix C gives the transformation matrices for the coordinate transformation routines in the Graphics Library.
- Appendix D discusses a feedback parser, which simplifies the use of the Geometry Engines in feedback mode.
- Appendix E provides manual pages for processes in the window manager environment.
- Appendix F provides the manual pages for the graphics labs in the *IRIS Programming Tutorial.*
- Appendix G discusses fast immediate mode macros, which speed execution of graphics code.
- Appendix H describes the Image Library, which is a library of routines that manipulate blocks of pixels.

1.3 Documentation Conventions

You can use the Graphics Library with three programming languages: C, FORTRAN, and Pascal. The C specification is given first, the FORTRAN second, and the Pascal third. For example:

```
move(x, y, z)
Coord x, y, z;

subroutine move(x, y, z)
real x, y, z

procedure move(x, y, a: Coord);
```

In the text, routines are printed in typewriter font, e.g., `move` and arguments and arrays are printed in italics, e.g., *x, y, z*. Italics are also used for words and expressions when they are defined in the text.

Each routine has a root name, such as `move`. The default world coordinate system is 3D with floating point coordinates. Suffixes are added to some routines to indicate 2D, integer (24 bits), and short integer (16 bits) arguments. Here are examples of the `move` variations:

```
move (1.0, 2.0, 3.0)      move2(1.0, 2.0)
movei (1, 2, 3)           move2i(1, 2)
moves(1, 2, 3)            move2s(1, 2)
```

The routine names are unique to six characters to conform to standard FORTRAN naming conventions. Routines are referred to by their full C or Pascal name in the text.

While FORTRAN programs are restricted to a small set of predefined data types, C and Pascal allow user-defined data types. Data types have been defined wherever it improves readability and reliability of the code. Appendix A gives the data type definitions used for the C and Pascal libraries.

Most important constants have been given symbolic names, such as XMAXSCREEN. Their values can be found in Appendix A. Other constants are often given in hexadecimal. The C syntax is used: the hexadecimal digits are preceded by "0x".

Sample code and programs are written in C and FORTRAN.

1.4 Related Publications

- *IRIS Programming Tutorial, C Edition*
- *IRIS Programming Tutorial, FORTRAN Edition*
- *IRIS Series 3000 Owner's Guide*
- *Getting Started with Your IRIS Workstation*
- *UNIX Programmer's Manual, Vol. IIA and Vol. IIB*
- *Learning to Debug with* edge, *C Edition*
- *Learning to Debug with* edge, *FORTRAN Edition*

2. Global State Attributes

This chapter introduces you to the IRIS programming environment. It describes the first steps you need to program on the IRIS:

- initializing an IRIS program
- exiting an IRIS program
- changing the global state attributes, i.e., the software and hardware environment

Initializing a program means telling the system to activate the software and hardware environment in which a program will run. Use `winopen` to initialize your program when running the window manager. Use `ginit` or `gbegin` when the window manager is not running.

Global state attributes are options that specify modes, linestyle, window specifications, and hardware requirements. Unless you specify otherwise, the global state attributes use their default values. (See Tables 2-1 and 2-2 below for the default values of the global state attributes.)

2.1 Initialization

The first Graphics Library routine in every IRIS program that is not running under the window manager is `ginit` or `gbegin`. These routines initialize the hardware, allocate memory for symbol tables and display list objects, and set up default values for global state attributes (See Tables 2-1 and 2-2). They have no arguments and should be called only once (before any other Graphics Library routine). When you are running the window manager, use `winopen` to perform the same operations. (See *Using mex, the IRIS Window Manager,* Chapter 2.)

Attribute	Initial Value	Section #
available bitplanes	all bitplanes[1]	6.1
blinking	turned off	6.2
color	undefined	6.3
color map mode	one map	6.2
cursor	0 (arrow)	6.4
depthcue mode	off	13.2
display mode	single buffer	6.1
font	0 [2]	5.3
linestyle	0 (solid)	5.1
linestyle backup	off	5.1
linewidth	1 pixel	5.1
lsrepeat	1	5.1
pattern	0 (solid)	5.2
picking size	10×10 pixels	9.2
reset linestyle	on	5.1
RGB color	undefined	6.3
RGB writemask	undefined	6.3
shaderange	0,7,0,1023	13.2
viewport	entire screen	4.4
writemask	all bitplanes enabled[1]	6.3
z-buffer mode	off	12.1

Table 2-1. Initial Values of Global State Attributes

1. If there are more than 3 bitplane boards installed, there are 12 displayable bitplanes.
2. Rasterfont 0 is a Helvetica-like type font.

Index	Name	RGB Value		
		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
all others	unnamed		undefined	

Table 2-2. Default Color Map Values

ginit

`ginit` performs the initialization functions and puts the default values into the color map (see Table 2-2) .

```
ginit()

subroutine ginit

procedure ginit;
```

Note: Under the window manager, use `winopen` instead of `ginit` or `gbegin`.

gbegin

`gbegin` performs all the same tasks as `ginit`, except it does not alter the color map.

```
gbegin()

subroutine gbegin

procedure gbegin;
```

greset

`greset` returns the global state attributes to their initial values. You can call `greset` at any time. Table 2-1 lists the global state attributes and their default values. It initializes the color map to the values shown in Table 2-2.

`greset` also performs the following tasks:

- sets up a 2-D orthographic projection transformation that maps user-defined coordinates to the entire area of the screen (see Chapter 4, Coordinate Transformations, Section 4.3).
- turns on the cursor and ties it to MOUSEX and MOUSEY (see Chapter 6, Display and Color Modes, Section 6.4 and Chapter 7, Input/Output Routines, Section 7.3).
- unqueues each button, each valuator, and the keyboard (see Chapter 7, Section 7.3).
- changes all buttons to FALSE (see Chapter 7).
- sets each valuator (except MOUSEY) to XMAXSCREEN/2, with a range of 0 to XMAXSCREEN (see Chapter 7).
- sets MOUSEY to YMAXSCREEN/2, with range 0 to YMAXSCREEN (see Chapter 7).

These tasks are discussed in detail throughout the *IRIS Graphics Programming Guide.*

```
greset()

subroutine greset

procedure greset;
```

gflush

When you use an IRIS terminal, the communications software buffers most graphics routines at the host for efficient block transfer of data from the host to the IRIS. `gflush` delivers all buffered but untransmitted graphics data to the IRIS. Certain graphics routines (notably those that return values) flush the host buffer when they execute. On the IRIS workstation, `gflush` does nothing.

```
gflush()
```

```
subroutine gflush
```

```
procedure gflush;
```

setslowcom

You use `setslowcom` on an IRIS terminal. `setslowcom` sends data in 6 bits per byte as certain connections require, e.g., RS232 connections. It has no effect on a workstation.

```
Boolean setslowcom()
```

```
logical function setslo()
```

```
function setslowcom(): Boolean;
```

setfastcom

You use `setfastcom` on an IRIS terminal. `setfastcom` sends data in 8 bits per byte as required by certain connections, e.g., some ethernet protocalls. It has no effect on a workstation.

```
Boolean setfastcom()
```

```
logical function setfas()
```

```
function setfastcom(): Boolean;
```

gexit

`gexit` is the final graphics routine in an IRIS program. `gexit` flushes communication buffers and waits for the graphics pipeline to empty.

```
gexit()
```

```
subroutine gexit
```

```
procedure gexit;
```

setmonitor

`setmonitor` selects one of five types of monitors: HZ30 = 30Hz interlaced, HZ50 = 50Hz noninterlaced, HZ60 = 60Hz noninterlaced, NTSC = NTSC (television standard encoding), and PAL, (European television encoding).

```
setmonitor(type)
short type;
```

```
subroutine setmon(type)
integer*4 type
```

```
procedure setmonitor(type: longint);
```

Note: For more information on using the Graphics Library subroutines with video options, see the *IRIS Owner's Guide*, Section 8.3, Using the Graphics Library with Video Options.

getmonitor

`getmonitor` returns the current monitor type.

```
long getmonitor()
```

```
integer*4 function getmon()
```

```
function getmonitor: longint;
```

getothermonitor

`getothermonitor` returns the nondisplayed monitor type. It complements `getmonitor`.

```
long getothermonitor()

integer*4 function getoth()

function getothermonitor: longint;
```

2.2 Saving Global State Attributes

pushattributes

`pushattributes` saves the current global state. The IRIS maintains a stack of global state attributes. `pushattributes` copies each of the following attributes onto the stack: color or RGB color, writemask or RGB writemask, font, linestyle, linestyle backup, reset linestyle, linewidth, pattern, and front and back buffer mode. Chapter 5, Linestyles, Patterns, and Fonts, and Chapter 6, Display and Color Modes, discuss these attributes.

```
pushattributes()

subroutine pushat

procedure pushattributes;
```

popattributes

`popattributes` restores the most recently saved values of the global state attributes.

```
popattributes()

subroutine popatt

procedure popattributes;
```

3. Drawing Routines

The IRIS Graphics Library includes routines for drawing points, lines, rectangles, polygons, circles, arcs, curves, and surfaces. It also includes routines for shading surfaces, for drawing text strings, and for writing and reading pixels. (See the *IRIS Programming Tutorial*, Chapter 3, for more information.) The drawing routines have variations:

- the number of dimensions in which the routine draws—two dimensions (2D) or three dimensions (3D)
- the type of numbers the routine uses in positioning arguments—integers (24 bits), short integers (16 bits), or floating point values
- the type of image the routine draws—outlined or filled

Points, lines, polygons, and text can be positioned in 2D or 3D. You can define rectangles, circles, and arcs in 2D, although you can translate them to 3D using the modeling routines described in Chapter 4, Coordinate Transformations. You always specify curves in 3D, although the z coordinate can be zero. You also specify surface patches in 3D.

The numbers you use for your argument can be integers, short integers, or floating point values. Routines that use floating point arguments are expressed in their original forms, e.g., `move`. Routines that use integers end in the letter *i*, e.g., `movei`. Routines that use short integers end in the letter *s*, e.g., `moves`.

Most routines that draw filled objects end in the letter *f*, e.g., `arcf`.

3.1 Current Drawing Positions

The IRIS maintains two current drawing positions that determine where drawing takes place when a drawing routine is called.

The *current graphics position* is a 3-D position expressed in floating point coordinates. All drawing routines, except `charstr`, `writepixels`, and `clear` update it. The current graphics position is set and used in sequences of point, line, and polygon routines. You can combine these routines with attribute-setting routines. (See Chapter 5, Linestyles, Patterns, and Fonts and Chapter 6, Display and Color Modes, for a discussion of attributes.) Other routines usually do not affect the graphics position.

The *current character position* is a 2-D position expressed in screen or window coordinates. It is set and used in sequences of `cmov`, `charstr`, and the four routines that read and write pixels.

getgpos

`getgpos` returns the current graphics position, after transformation and before clipping and scaling. (Use the fourth coordinate, *fw*, for clipping and perspective division.)

```
getgpos(fx, fy, fz, fw)
Coord *fx, *fy, *fz, *fw;

subroutine getgpo(fx, fy, fz, fw)
real fx, fy, fz, fw

procedure getgpos(var fx, fy, fz, fw: Coord);
```

getcpos

`getcpos` returns the current character position.

```
getcpos(ix, iy)
Screencoord *ix, *iy;

subroutine getcpo(ix, iy)
integer*2 ix, iy

procedure getcpos(var ix, iy: Screencord);
```

3.2 Clearing the Viewport

clear

`clear` sets the screen area within the current viewport to the current color using the current writemask and pattern. (See Chapter 4, Section 4.4 for a discussion of viewports; see Chapter 5, Linestyles, Patterns, and Fonts and Chapter 6, Display and Color Modes, for descriptions of the color, writemask, and pattern attributes.) `clear` leaves the current graphics position and the current character position undefined.

```
clear()

subroutine clear

procedure clear;
```

3.3 Points

pnt

`pnt` draws a point at a specified position. If the point is visible in the window, it appears as one pixel using the current color. `pnt` updates the current graphics position to its location.

```
pnt(x, y, z)
Coord x, y, z;

subroutine pnt(x, y, z)
real x, y, z

procedure pnt(x, y, z: Coord);
```

The following program draws 100 points in a square area of the window:

■ C Program: SQUARE DRAWN WITH POINTS

```
#include "gl.h"

main()
{
    int i,j;

    ginit();
    cursoff(); /* turn off cursor so it */
        /* doesn't interfere with drawing */
    color(BLACK); /* make BLACK the current drawing color */
    clear(); /* clear the screen (to black) */
    color(BLUE); /* make BLUE the current drawing color */

    for (i=0; i<10; i=i+1) {
        for (j=0; j<10; j=j+1)
            pnti(i*5,j*5,0);
        }
    sleep(5);
    gexit();
}
```

■ FORTRAN Program: SQUARE DRAWN WITH POINTS

```
    #INCLUDE /usr/include/fgl.h
    #INCLUDE /usr/include/fdevice.h

    INTEGER I , J

    CALL GINIT
    CALL CURSOF
    CALL COLOR(BLACK)
    CALL CLEAR
    CALL COLOR(BLUE)
    DO 10 I = 0 , 9
       DO 20 J = 0 , 9
        CALL PNTI(I*5,J*5,0)
 20    CONTINUE
 10 CONTINUE
```

```
111   CONTINUE
    IF (.NOT. GETBUT(RIGHTM))GO TO 111
    call color(BLACK)
    call clear
    CALL GEXIT
    STOP
    END
```

3.4 Lines

`move` and `draw` draw lines.

move

`move` changes the current graphics position to the specified position.

```
move(x, y, z)
Coord x, y, z;

subroutine move(x, y, z)
real x, y, z

procedure move(x, y, z: Coord);
```

draw

`draw` draws a line from the current graphics position to the point (*x*, *y*, *z*). The appearance of the line is determined by the current linestyle, linewidth, linestyle repeat, linestyle backup, color, and writemask (see Chapter 5, Linestyles, Patterns, and Fonts, and Chapter 6, Display and Color Modes). `draw` updates the current graphics position to the point (*x*, *y*, *z*).

```
draw(x, y, z)
Coord x, y, z;

subroutine draw(x, y, z)
real x, y, z

procedure draw(x, y, z: Coord);
```

The following program draws the outline of a blue box on the screen using `move` and `draw`.

■ C Program: BLUE BOX

```
#include "gl.h"

main()
{
    ginit();
    cursoff();
    color(BLACK);
    clear();
    color(BLUE);

    move2i(200,200);
    draw2i(200,300);
    draw2i(300,300);
    draw2i(300,200);
    draw2i(200,200);
    sleep(10); /* sleep for ten seconds before returning
        to textport */
    gexit();
}
```

■ FORTRAN Program: BLUE BOX

```
#INCLUDE /usr/include/fgl.h
#INCLUDE /usr/include/fdevice.h

  call ginit
  call cursof
  call color(BLACK)
  call clear
  call color(BLUE)

  call move2i(200,200)
  call draw2i(200,300)
  call draw2i(300,300)
  call draw2i(300,200)
  call draw2i(200,200)
```

```
5     continue
      if (.not. getbut(RIGHTM))go to 5
      call color(BLACK)
      call clear
      call gexit
      stop
      end
```

3.4.1 Relative Drawing

The *relative drawing* routines, `rmv` and `rdr`, interpret their arguments using the current graphics position as an origin.

rmv

`rmv(a,b,c)` changes the current graphics position from (*x*, *y*, *z*) to (*x*+a, *y*+b, *z*+c). It moves (without drawing) the graphics position the amount specified, relative to its current value.

```
rmv(dx, dy, dz)
Coord dx, dy, dz;

subroutine rmv(dx, dy, dz)
real dx, dy, dz

procedure rmv(dx, dy, dz: Coord);
```

rdr

`rdr(a,b,c)` draws a line from the current graphics position (*x*, *y*, *z*) to (*x*+a, *y*+b, *z*+c), and sets the current graphics position to (*x*+a, *y*+b, *z*+c).

```
rdr(dx, dy, dz)
Coord dx, dy, dz;

subroutine rdr(dx, dy, dz)
real dx, dy, dz

procedure rdr(dx, dy, dz: Coord);
```

3.5 Rectangles

Two points specifying opposite corners determine a rectangle. The sides of the rectangle are parallel to the *x* and *y* axes; the *z* coordinate is zero.

rect

`rect` draws the outline of a rectangle; `rectf` draws a filled rectangle. Since a rectangle is a 2-D shape, these routines take only 2-D arguments and set the *z* coordinate to zero.

`rect` takes four arguments: *x1*, *y1*, *x2*, and *y2*. A rectangle is outlined by four line segments using the current linestyle, linewidth, linestyle repeat, linestyle backup, color, and writemask (Figure 3-1).

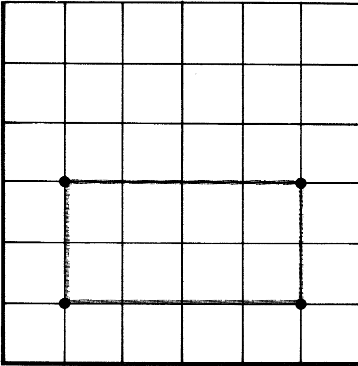
```
rect(x1, y1, x2, y2)
Coord x1, y1, x2, y2;

subroutine rect(x1, y1, x2, y2)
real x1, y1, x2, y2

procedure rect(x1, y1, x2, y2: Coord);
```

rectf

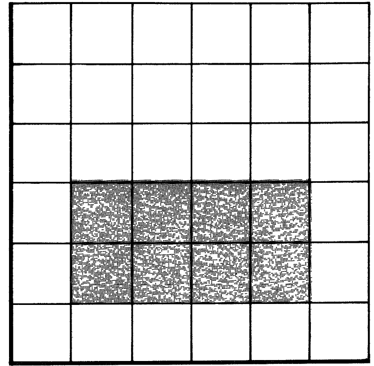
`rectf` takes the same arguments as `rect` and produces a filled rectangular region using the current pattern, color, and writemask (Figure 3-1). Both `rect` and `rectf` set the current graphics position to *(x1, y1)*.

You must specify the lower-left and upper-right corners of the rectangle in backface mode.

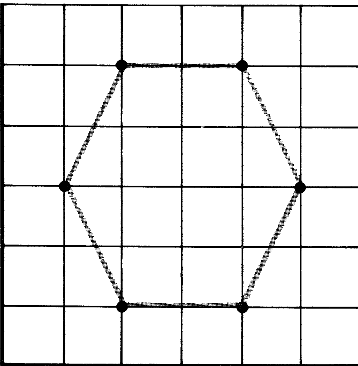
```
rectf(x1, y1, x2, y2)
Coord x1, y1, x2, y2;

subroutine rectf(x1, y1, x2, y2)
real x1, y1, x2, y2

procedure rectf(x1, y1, x2, y2: Coord);
```



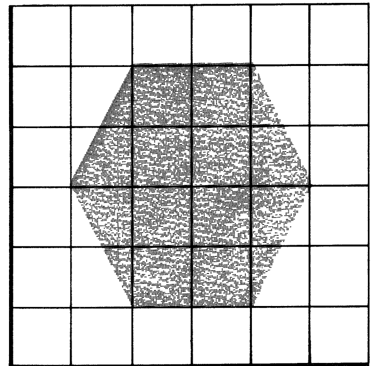
recti (1,1,5,3)



rectfi (1,1,5,3)



poly2i (6,parray)



polf2i (6,parray)

```
static Icoord parray [6][2]={
                                {2,1},
                                {1,3},
                                {2,5},
                                {4,5},
                                {5,3},
                                {4,1}
                                    };
```

An array of object space points defines a polygon. You can draw rectangles and polygons as outlined or filled areas. Specify rectangles by opposite corners.

Figure 3-1. Filled and Unfilled Rectangles and Polygons

rectcopy

`rectcopy` copies a rectangular array of pixels defined in screen coordinates (relative to the lower-left corner of the window under the window manager) to another position on the screen. The lower-left corner of the new rectangle is defined by *newx* and *newy*.

```
rectcopy(x1, y1, x2, y2, newx, newy)
Screencoord x1, y1, x2, y2, newx, newy;

subroutine rectco(x1, y1, x2, y2, newx, newy)
integer*4 x1, y1, x2, y2, newx, newy

procedure rectcopy(x1, y1, x2, y2, newx, newy: Screencoord);
```

3.6 Polygons

`poly` outlines a polygonal area; `polf` fills a polygonal area. The specified array of points represents a polygon; the first and last points connect to close the polygon. Figure 3-1 shows identical polygons: one drawn with `poly` showing a solid linestyle and one drawn with `polf` showing a solid pattern. `poly` and `polf` set the current graphics position to the first point in *parray*.

poly

`poly` outlines a polygon. It takes two arguments: the number of points in the polygon *(n)* and the array of coordinates *(parray)*. You can express points in 2D or 3D, using integers, shorts, or floating point numbers. The system draws the polygon using the current linestyle, linestyle repeat, linestyle backup, linewidth, color, and writemask. All polygons must be convex.

```
poly(n, parray)
long n;
Coord parray[][3];

subroutine poly(n, parray)
integer*4 n
real parray(3,n)

procedure poly(n: longint; var parray: Coord);
```

polf

polf is the same as poly, except it fills a polygon using the current pattern, color, and writemask. All filled polygons must be convex. The system does not report any errors if you specify concave polygons, although they produce unpredictable results.

```
polf(n, parray)
long n;
Coord parray[][3];

subroutine polf(n, parray)
integer*4 n
real parray(3,n)

procedure polf(n: longint; var parray: Coord);
```

You can also draw filled polygons by specifying one vertex at a time. pmv specifies the first point in a polygon. A sequence of pdr routines then specifies a sequence of subsequent points in the polygon. pclos fills the polygon specified by the preceding pmv and pdr routines using the current color, writemask, and pattern. rpdv and rpdr are relative versions of pmv and pmr.

Note: Do not issue any Graphics Library routines other than pdr, rpdr, and setshade between the initial pmv or rpmv and the final pclos.

pmv

pmv moves to the starting point of a filled polygon.

```
pmv(x, y, z)
Coord x, y, z;

subroutine pmv(x, y, z)
real x, y, z

procedure pmv(x, y, z: Coord);
```

rpmv

`rpmv` specifies a relative move to the starting point of a filled polygon, using the current graphics position as the origin.

```
rpmv(dx, dy, dz)
Coord dx, dy, dz;

subroutine rpmv(dx, dy, dz)
real dx, dy, dz

procedure rpmv(dx, dy, dz: Coord);
```

pdr

`pdr` specifies the next point in a filled polygon.

```
pdr(x, y, z)
Coord x, y, z;

subroutine pdr(x, y, z)
real x, y, z

procedure pdr(x, y, z: Coord);
```

rpdr

`rpdr` specifies the next point in a filled polygon, using the previous point (the current graphics position) as the origin.

```
rpdr(dx, dy, dz)
Coord dx, dy, dz;

subroutine rpdr(dx, dy, dz)
real dx, dy, dz

procedure rpdr(dx, dy, dz: Coord);
```

pclos

`pclos` fills the polygon specified by the preceding sequence of `pmv`, `rpmv`, `pdr`, or `rpdr` routines.

```
pclos()

subroutine pclos

procedure pclos;
```

Note: Do not issue any Graphics Library routines other than `pdr`, `rpdr`, and `setshade` between the initial `pmv` or `rpmv` and the final `pclos`.

Do not confuse `pclos` with the UNIX system call *pclose*, which closes a UNIX pipe.

3.7 Circles and Arcs

3.7.1 Circles

`circ` outlines a circle and `circf` draws filled circles. A circle is defined by a center point *(x,y)* and a radius *(radius)* in the *x-y* plane, with $z = 0$. Since a circle is a 2-D shape, these routines have only 2-D forms. (Note that circles rotated outside of the 2-D *x-y* plane appear as ellipses.)

circ

`circ` outlines a circle. The circle has a center point *(x,y)* and a radius *(radius)*, which are specified in world coordinates. The IRIS draws circles using the current linestyle, linewidth, linestyle repeat, linestyle backup, color, and writemask.

```
circ(x, y, radius)
Coord x, y, radius;

subroutine circ(x, y, radius)
real x, y, radius

procedure circ(x, y, radius: Coord);
```

circf

`circf` draws filled circles. The IRIS uses the current color, pattern, and writemask to fill the circle with center *(x,y)* and radius *(radius)*. `circ` and `circf` set the current graphics position to *(x+radius, y)*.

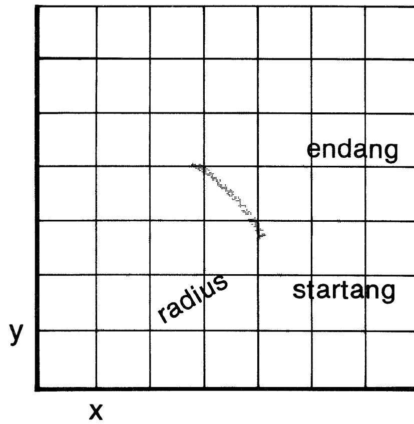
```
circf(x, y, radius)
Coord x, y, radius;

subroutine circf(x, y, radius)
real x, y, radius

procedure circf(x, y, radius: Coord);
```

3.7.2 Arcs

`arc` outlines a circular arc and `arcf` draws a filled circular arc. Arcs are defined by a center point *(x,y)*, a radius *(radius)*, a starting angle *(startang)*, and an ending angle *(endang)*. The angles are measured from the *x* axis and are specified in integral tenths of degrees; positive angles describe counterclockwise rotations. Since an arc is a 2-D shape, these routines have only 2-D forms. The IRIS draws an arc using the current color, linestyle, linestyle repeat, linestyle backup, linewidth, and writemask. Figure 3-2 shows an arc and the parameters that define it.

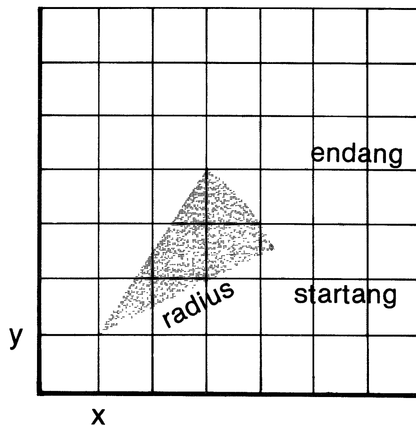
`arc` and `arcf` leave the current graphics position undefined.

(a)



arci(x,y,radius,startang,endang);

(b)



arcfi(x,y,radius,startang,endang);

A center point, radius, start angle, and end angle define circular arcs. They are drawn counterclockwise in the x-y plane, with angles measured from the x-axis.

Figure 3-2. Arcs

arc

`arc` outlines a circular arc.

```
arc(x, y, radius, startang, endang)
Coord x, y, radius;
Angle startang, endang;

subroutine arc(x, y, radius, stang, endang)
real x, y, radius
integer*4 stang, endang

procedure arc(x, y, radius: Coord; startang,
     endang: Angle);
```

arcf

`arcf` draws filled arcs using the current pattern, color, and writemask (Figure 3-2). `arc` and `arcf` set the current graphics position to the endpoint of the arc.

```
arcf(x, y, radius, startang, endang)
Coord x, y, radius;
Angle startang, endang;

subroutine arcf(x, y, radius, stang, endang)
real x, y, radius
integer*4 stang, endang

procedure arcf(x, y, radius: Coord; startang,
     endang: Angle);
```

3.8 Text

Use `cmov` and `charstr` to create text. `cmov` determines where the system draws text on the screen, and `charstr` draws a string of characters.

cmov

The current character position (see Section 3.1) determines where the IRIS draws text on the screen. `cmov` moves the current character position to a specified point (as `move` sets the current line drawing position). *x*, *y*, and *z* are integers, shorts, or real numbers in 2D or 3D, which specify a point in

world coordinates. `cmov` transforms the world coordinates into window coordinates, which become the new character position. `cmov` does not affect the current graphics position.

```
cmov(x, y, z)
Coord x, y, z;

subroutine cmov(x, y, z)
real x, y, z

procedure cmov(x, y, z: Coord);
```

charstr

`charstr` draws a string of raster characters. The origin of the first character in the string is the current character position. After the system draws the string, it updates the current character position to the pixel to the right of the last character in the string. (Character strings are null-terminated in C.) The text string is drawn in the current font and color. (See Chapter 5, Linestyles, Patterns, and Fonts, Section 5.3, for a discussion of fonts.)

```
charstr(str)
String str;

subroutine charst(str, length)
character*(*) str
integer*4 length

procedure charstr(str: pstring128);
```

If the origin of a character string lies outside the viewport, none of the characters in the string are drawn. If the origin is inside the viewport, the characters are individually clipped to the screenmask. (Chapter 4, Coordinate Transformations, discusses viewports and screenmasks.) The screenmask is normally set to the same size as the viewport, although it can be set smaller than the viewport to enable two kinds of clipping. *Gross clipping* removes all strings that start outside the viewport (Figure 4-8). *Fine clipping* trims individual characters to the screenmask.

The following example draws two lines of text. The program assumes the current font is less than 12 pixels high.

■ C Program: TEXT

```
#include "gl.h"

main()
v{
    ginit();
    cursoff(); /* turn the cursor off so it won't interfere
        with the text */
    color(BLACK);
    clear();
    color(RED);
    cmov2i(300,380);
    charstr("The first line is drawn ");
    charstr("in two parts. ");
    cmov2i(300, 368);
    charstr("This line is 12 pixels lower. ");
    sleep(5); /* pause for five seconds before returning
        to textport */
    curson(); /* turn the cursor back on */
    gexit();
}
```

■ FORTRAN Program: TEXT

```
    #INCLUDE /usr/include/fgl.h
    #INCLUDE /usr/include/fdevice.h

    CALL GINIT
    call cursof
    CALL COLOR(BLACK)
    CALL CLEAR
    CALL COLOR(RED)
    CALL CMOV2I(300,380)
    CALL CHARST('The first line is drawn ',24)
    CALL CHARST('in two parts. ',14)
    CALL CMOV2I(300,368)
    CALL CHARST('This line is 12 pixels lower. ',30)
 99 continue
    if(.not. getbut(RIGHTM))go to 99
```

```
call color(BLACK)
    call clear
call curson
CALL GEXIT
STOP
END
```

3.9 Writing and Reading Pixels

`writepixels` and `writeRGB` paint one or more pixels on the screen. The routines specify the number of pixels to paint and a color for each pixel. The starting location is the current character position. The system updates that position to the pixel that follows the last painted pixel. The current character position becomes undefined if the next pixel position is greater than XMAXSCREEN. The system paints pixels from left to right and clips them to the current screenmask (see Chapter 4, Section 4.4). These routines do not automatically wrap from one line to the next.

writepixels

`writepixels` paints a row of pixels on the screen in color map mode. (See Chapter 6, Section 6.2, for an explanation of color maps.) *n* specifies the number of pixels to paint and *colors* specifies a color for each pixel. `writepixels` does not automatically wrap from one line to the next. You can use it in single buffer and double buffer modes.

Note: For higher level constructs built with `writepixels`, see Appendix H, Image Library.

```
writepixels(n, colors)
short n;
Colorindex colors[];

subroutine writep(n, colors)
integer*4 n
integer*2 colors(n)

procedure writepixels(n: longint; var colors: Colorindex);
```

writeRGB

`writeRGB` paints a row of pixels on the screen in RGB mode. *n* specifies the number of pixels to paint; *red, green, blue* specify arrays of colors for each pixel.

`writeRGB` does not automatically wrap from one line to the next. It supplies a 24-bit RGB value (8 bits each for *red, green,* and *blue*) for each pixel. `writeRGB` writes the 24-bit RGB value directly into the bitplanes. (See Chapter 6, Section 6.1, for an explanation of RGB mode.)

```
writeRGB(n, red, green, blue)
short n;
RGBvalue red[], green[], blue[];

subroutine writeR(n, red, green, blue)
integer*4 n
character*(*) red, green, blue

procedure writeRGB(n: longint; var red, green, blue:
     RGBvalue);
```

readpixels

`readpixels` reads pixel values from the bitplanes in color map mode. It attempts to read up to *n* pixel values, starting from the current character position and moving along a single scan line (constant *y*) in the direction of increasing *x*. `readpixels` returns the number of pixels the system actually reads. In double buffer mode, the system reads the pixel values from the back buffer. The values of pixels read outside the current viewport are undefined. The IRIS updates the current character position to the pixel to the right of the last one read. The current character position is undefined if the new position is outside the screen. `readpixels` does not wrap to the next line of pixels when the current character position encounters the edge of the screen.

Note: For higher level constructs built with `readpixels`, see Appendix H, Image Library.

```
long readpixels(n, colors)
short n;
Colorindex colors[];

integer*4 function readpi(n, colors)
integer*4 n
integer*2 colors(n)

function readpixels(n: longint; var colors:
    Colorindex): longint;
```

readRGB

`readRGB` reads up to *n* pixel values from the bitplanes in RGB mode. They are read into the *red*, *green*, and *blue* arrays starting from the current character position along a single scan line (constant *y*) in the direction of increasing *x*. `readRGB` returns the number of pixels the system actually reads. The values of pixels read outside the current screen are undefined. `readRGB` updates the current character position to the pixel to the right of the last one read. The current character position is undefined if the new position is outside the screen.

```
long readRGB(n, red, green, blue)
short n;
RGBvalue red[], green[], blue[];

integer*4 function readRG(n, red, green, blue)
integer*4 n
character*(*) red, green, blue

function readRGB(n: longint; var red, green, blue:
    RGBvalue): longint;
```

4. Coordinate Transformations

The IRIS creates 3-D shapes that you can manipulate and view. You use coordinate transformations to move, scale, and rotate these shapes; you use viewing transformations to change your point of view; and you use projection routines to determine how you view the object.

There are four coordinate systems that perform coordinate transformations:

- Object coordinate system, which is the coordinates of the object itself, e.g., the coordinate system of a cube. It is independent of other coordinate systems.
- World coordinate system, which is the coordinates of the world in which the object exists. Each object's coordinates are relative to those of other objects in its world, e.g., each cube in a stack of three cubes has different world coordinates because it is relative to the position of the other cubes in world space.
- Eye coordinate system, which is the coordinate system in which your eye sits at the origin and looks down the -*z* axis.
- Screen coordinate system, which is the coordinate system in which the transformed object appears on the screen. The viewport determines the screen coordinates. When you use the window manager, it determines the screen coordinates within the window.

The relationship of the above transformations is analogous to the four basic steps in performing coordinate transformations of graphical objects:

1. Construct a graphical object using coordinates.
2. Transform the graphical object to the correct location within the world coordinate system. (Use modeling routines such as `rotate`, `translate`, or `scale`.)

3. Specify the eye's position and viewing direction to transform the eye to the origin and the object of interest along the -*z* axis. (Use viewing transformation routines such as `polarview` or `lookat`.)
4. Choose a projection to specify which region of the graphical object will appear on the screen. (Use projection routines such as `ortho`, `ortho2`, `window`, or `perspective`.)

Each coordinate transformation is incorporated into a transformation matrix, which reflects the cumulative effect of all transformations. The coordinates of every drawing routine are multiplied by the current transformation matrix. The current transformation matrix is the top matrix in a stack of 32 4x4 floating point matrices. Eight matrices are actually in hardware; if you use eight or fewer, you can get better performance. (See the *IRIS Programming Tutorial*, Chapters 6 and 7, for detailed information on matrices.) There are five routines you can use to manipulate the matrix stack: `loadmatrix`, `getmatrix`, `multmatrix`, `pushmatrix`, and `popmatrix`.

4.1 Modeling Transformations

Each graphical object, or geometric model, is defined in its own coordinate system. You can manipulate the entire object using the modeling transformation routines: `rotate`, `rot`, `translate`, and `scale`. By combining or linking together drawing routines, you can create more complex modeling transformations that express relationships between different parts of a complex object.

rotate

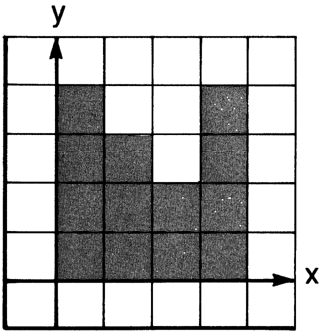
`rotate` rotates graphical objects; it specifies an angle and an axis of rotation. The angle is given in tenths of degrees according to the right-hand rule, which is as follows: as you look down the positive rotation axis to the origin, positive rotation is counterclockwise. A character, either *x*, *y*, or *z*, defines the axis of rotation. (The character can be upper- or lowercase.) For example, the object shown in Figure 4-1(a) is rotated 30 degrees with respect to the *y* axis in Figure 4-1(b). All objects drawn after `rotate` executes are rotated.

```
rotate(a, axis)
Angle a;
char axis;

subroutine rotate(a, axis)
integer*4 a
character axis

procedure rotate(a: longint; axis: longinit);
```

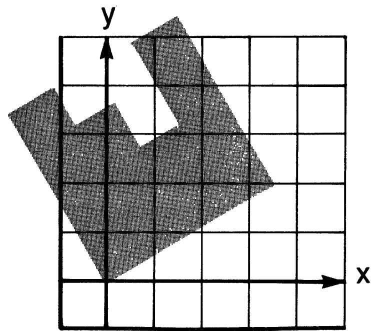
rot

`rot` is the same as `rotate`; it specifies the angle as a floating point value.

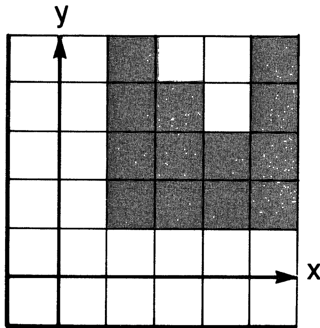
```
rot(a, axis)
float a;
char axis;

subroutine rot(a, axis)
real a
character axis

procedure rot(a: real; axis: longint);
```



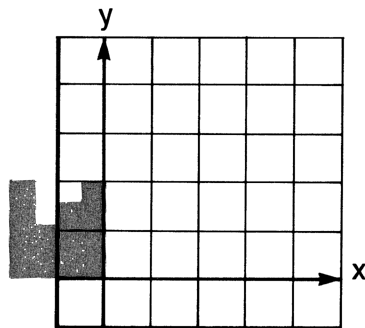
(a) original object at (0,0,0)



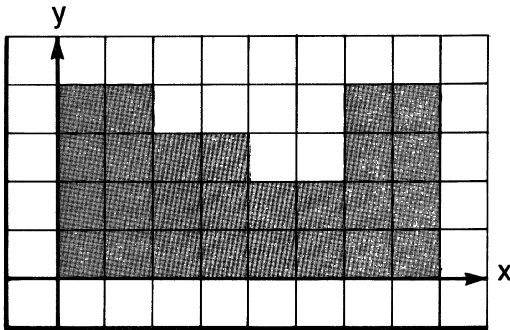
(b) rotate (300 ,'Z');



(c) translate (1.,1.,0.);



(d) scale (-.5,.5,1.);



(e) scale (2.,1.,1.);

The modeling routines are *rotate, translate,* and *scale.* The object shown in (a) is rotated in (b), translated in (c), and scaled in (d) and (e).

Figure 4-1. Modeling Routines

translate

`translate` moves the object origin to the point specified in the current object coordinate system. The object in Figure 4-1(a) is translated in Figure 4-1(c). All objects drawn after `translate` executes are translated.

```
translate(x, y, z)
Coord x, y, z;

subroutine transl(x, y, z)
real x, y, z

procedure translate(x, y, z: Coord);
```

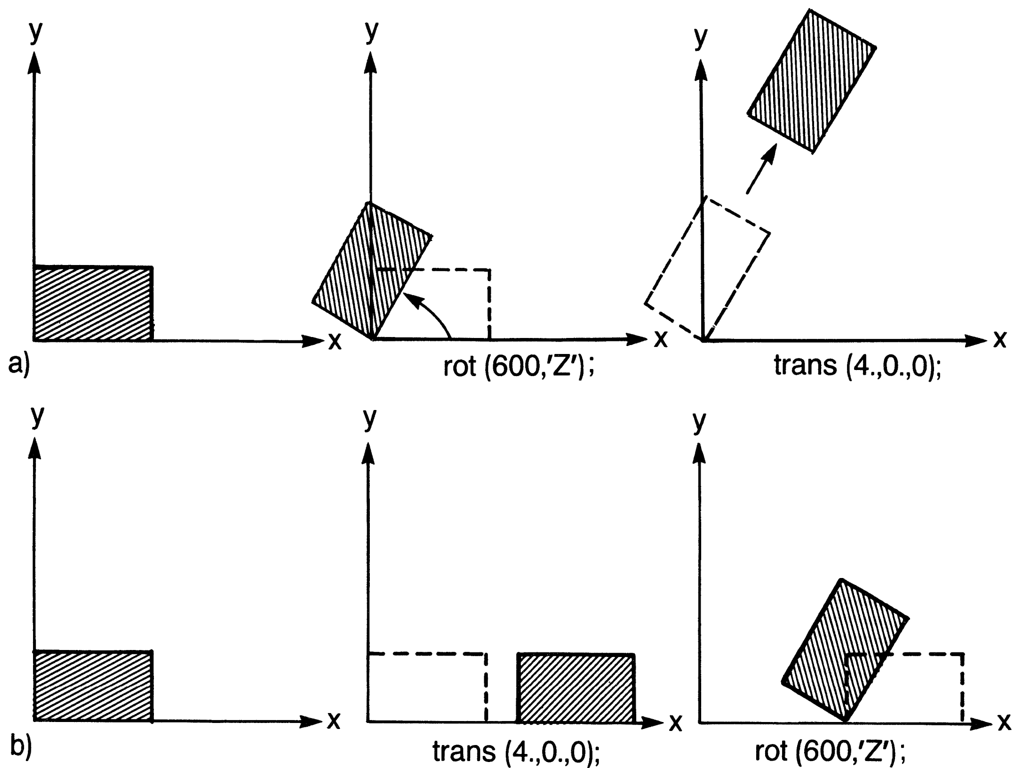
scale

`scale` shrinks, expands, and mirrors objects. Its three arguments *(x, y, z)* specify scaling in each of the three coordinate directions. Values with magnitude of 1 or more expand the object; values with magnitudes of less than 1 shrink it. Negative values cause mirroring.

All objects that are drawn after `scale` executes are scaled. The object shown in Figure 4-1(a) is shrunk to one-quarter of its original size and is mirrored about the *y* axis in Figure 4-1(d). It is scaled only in the *x* direction in Figure 4-1(e).

```
scale(x, y, z)
float x, y, z;

subroutine scale(x, y, z)
real x, y, z

procedure scale(x, y, z: real);
```

You can combine `rotate`, `rot`, `translate`, and `scale` to produce more complicated transformations. The order in which you apply these transformations is important. Figure 4-2 shows two different sequences of `translate` and `rotate`; each sequence has different results.



The modeling routines are not commutative: if you reverse the order of operations, you can get different results. (a) shows a rotation of 60 degrees about the origin followed by a translation of 4 degrees in the x-direction. (b) shows the same operations performed in the reverse order. Note that rotations are about the origin of the coordinate system.

Figure 4-2. Modeling Routine Order

4.2 Viewing Transformations

The viewing transformations place the viewer and the eye coordinate system in world space. In the process, they define the eye coordinate system. `polarview` and `lookat` define a right-hand world coordinate system with *x* to the right, *y* up, and *z* toward the viewer. You specify all rotations with integers in tenths of degrees. Rotations obey the right-hand rule. As the viewer looks down a positive axis to the origin, a positive rotation about an axis is counterclockwise. You can choose other world space orientations and create viewing transformations from the projection routines described in Section 4.3.

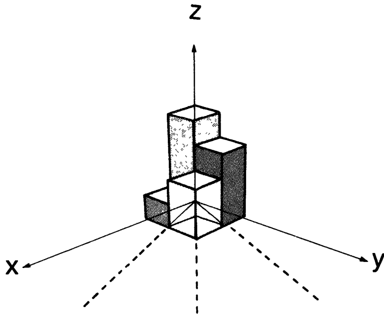
If no viewing transformation is specified, the eye is assumed to be at the origin looking down the -*z* axis. The viewing transformations `polarview` and `lookat` transform different eye points and viewing directions to this standard orientation.

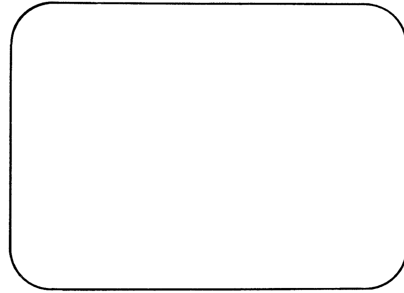
Note: A viewing transformation routine should always follow a projection routine.

polarview

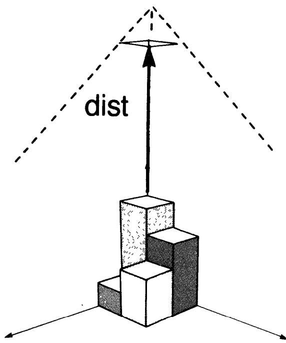
`polarview` defines the viewer's position in polar coordinates. The first three arguments, *dist*, *azim*, and *inc*, define a viewpoint. *dist* is the distance from the viewpoint to the world space origin. *azim* is the azimuthal angle in the *x*-*y* plane, measured from the *y* axis. *inc* is the incidence angle in the *y*-*z* plane, measured from the *z* axis. The line of sight is the line between the viewpoint and the world space origin. *twist* rotates the viewpoint around the line of sight using the right-hand rule. All angles are specified in tenths of degrees and are integers. Figure 4-3 shows examples of `polarview`.

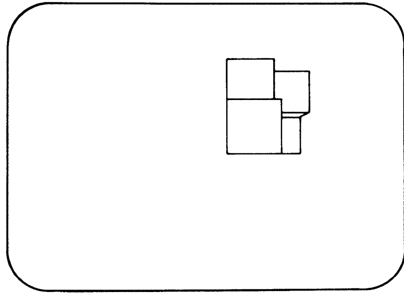
```
polarview(dist, azim, inc, twist)
Coord dist;
Angle azim, inc, twist;

subroutine polarv(dist, azim, inc, twist)
real dist
integer*4 azim, inc, twist

procedure polarview(dist: Coord; azim, inc, twist: longint);
```





polarview(0.,0,0,0);

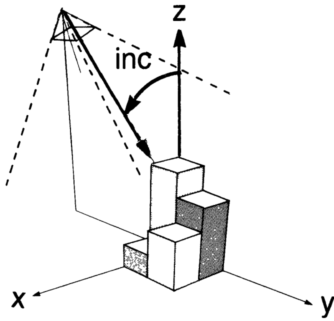


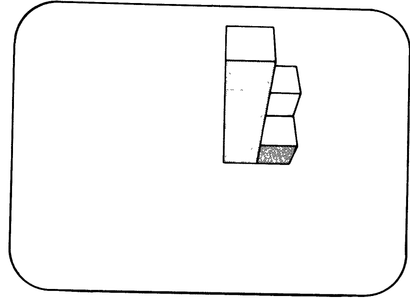


polarview(10.,0,0,0);

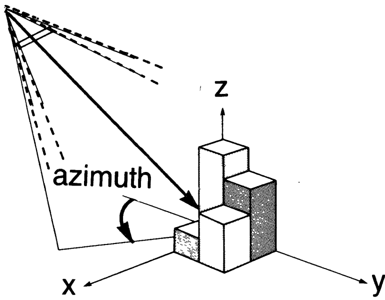
polarview has four arguments: the viewing distance from the origin, an incidence angle measured from the z-axis in the y-z plane, an azimuthal angle measured from the y-axis in the x-y plane, and a twist around the line of sight. Each frame shows the viewpoint and viewed image as additional arguments to *polarview* are supplied.

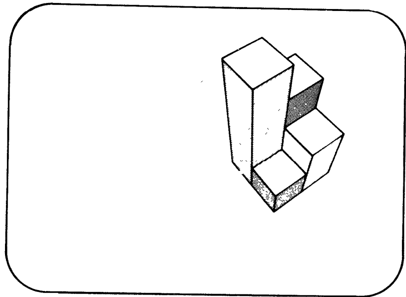
Figure 4-3. polarview



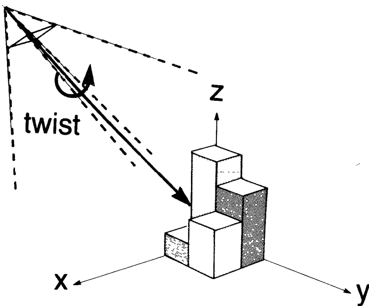


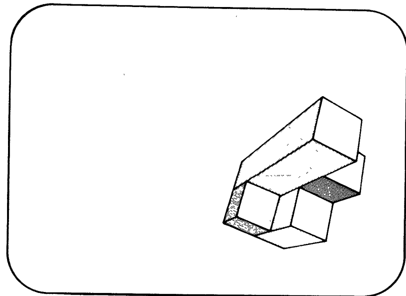
polarview(10.,0,300,0);





polarview(10.,600,300,0);





polarview(10.,600,300,450);

Figure 4-3. polarview (continued)

lookat

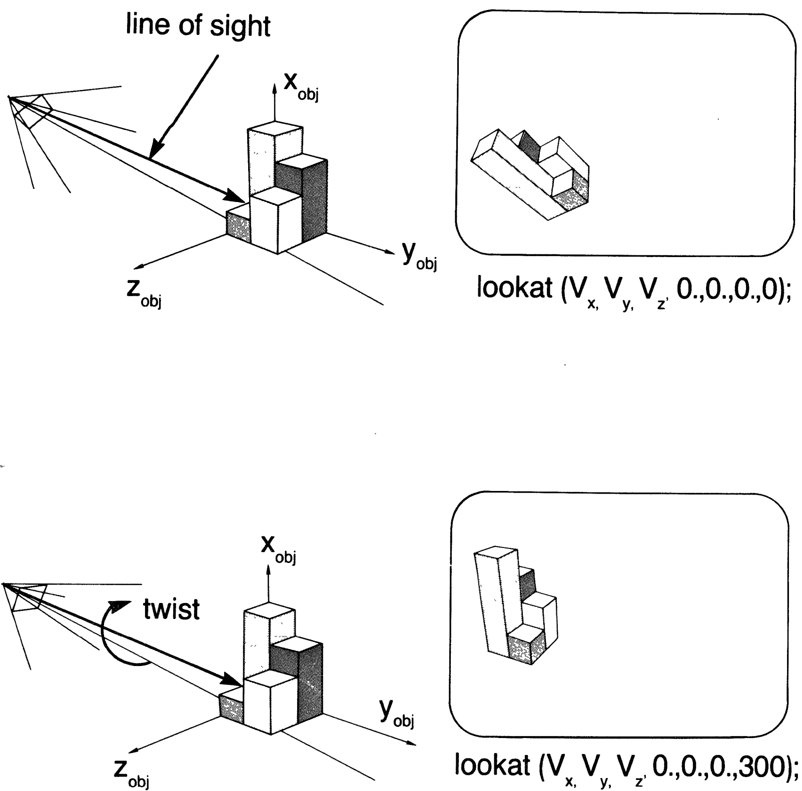
`lookat` defines a viewpoint and a reference point on the line of sight in world coordinates. The viewpoint is at *(vx, vy, vz)* and the reference point is at *(px, py, pz)*. These two points define the line of sight. *twist* measures right-hand rotation about the *z* axis in the eye coordinate system. Figure 4-4 illustrates `lookat`.

```
lookat(vx, vy, vz, px, py, pz, twist)
Coord vx, vy, vz, px, py, pz;
Angle twist;

subroutine lookat(vx, vy, vz, px, py, pz, twist)
real vx, vy, vz, px, py, pz
integer*4 twist

procedure lookat(vx, vy, vz, px, py, pz:
     Coord; twist: longint);
```

4.3 Projection Transformations

Projection transformations define the mapping from an eye coordinate system to the screen. The eye is placed at the origin looking down the -*z* axis. A viewport, which specifies a screen area to display the projected image, is associated with each projection transformation. Projection transformations also load the current matrix (see Section 4.5 for detailed information).

`perspective` and `window` specify perspective viewing pyramids into the world coordinate system and differ only in the method of defining the pyramid, as described below. `ortho` defines a 3-D viewing box (a rectangular parallelepiped) and `ortho2` defines a 2-D viewing rectangle for orthographic projections.



lookat defines a viewpoint, a reference point along the line of sight, and a twist angle. The top illustrations show the viewer and viewed image with no twist; twist is added to the lower illustrations.

Figure 4-4. lookat

perspective

`perspective` defines the viewing pyramid by indicating the field-of-view angle *(fovy)* in the *y* direction of the eye coordinate system, the aspect ratio *(aspect)* which determines the field of view in the *x* direction, and the location of the near and far clipping planes *(near* and *far)* in the *z* direction. The clipping planes are the boundaries of the viewing pyramid. (See the *IRIS Programming Tutorial*, Chapter 7, for detailed information.) You specify the aspect ratio as a ratio of *x* to *y*.

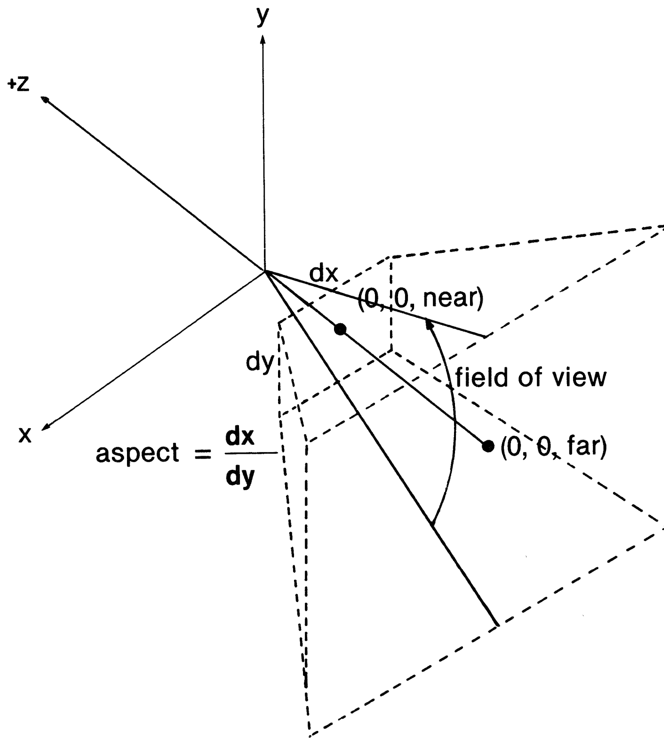
In general, the aspect ratio given in `perspective` should match the aspect ratio of the associated viewport. For example, *aspect* = 2 means the viewer's angle of view is twice as wide in *x* as in *y*. If the viewport is also twice as wide as it is tall, it displays the image without distortion. The arguments *near* and *far* are distances from the viewer to the near and far clipping planes, and are always positive. Figure 4-5 illustrates `perspective`.

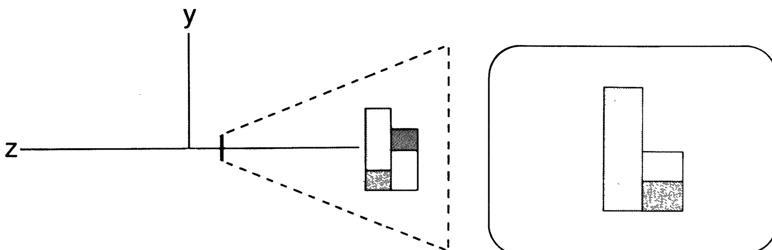
```
perspective(fovy, aspect, near, far)
Angle fovy;
float aspect;
Coord near, far;

subroutine perspe(fovy, aspect, near, far)
integer*4 fovy
real aspect, near, far

procedure perspective(fovy: longint; aspect: real;
     near, far: Coord);
```

window

`window` specifies the position and size of the rectangular viewing frustum closest to the eye (in the near clipping plane), and the location of the far clipping plane. `window` projects the image onto the screen with perspective. See Figure 4-6.

```
window(left, right, bottom, top, near, far)
Coord left, right, bottom, top, near, far;

subroutine window(left, right, bottom, top, near, far)
real left, right, bottom, top, near, far

procedure window(left, right, bottom, top, near, far:
      Coord);
```

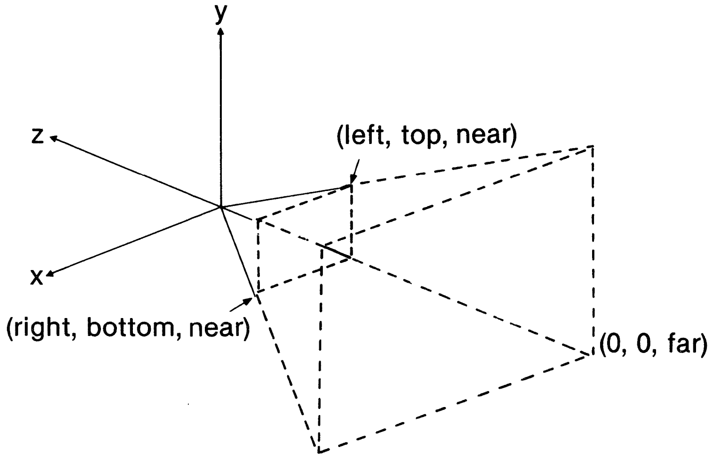


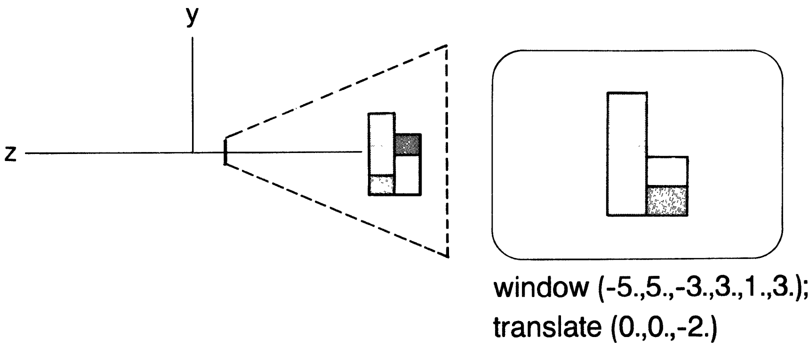


perspective (400,1.3,1.,3.);
translate (0.,0.,-2.);

perspective places the eye at world origin looking down the -z axis. It defines a field of view, an aspect ratio, and near and far clipping planes relative to the eye.

Figure 4-5. perspective





window defines a viewing window in the x-y plane looking down the -z axis. A perspective view of the image is projected onto the window.

Figure 4-6. window

ortho

`ortho` defines a box-shaped enclosure in the eye coordinate system. *left*, *right*, *bottom*, and *top* define the *x* and *y* clipping planes. *near* and *far* are distances along the line of sight and can be negative. In other words, the *z* clipping planes are located at z = -near and z = -far. Figure 4-7 shows an example of a 3-D orthographic projection.

```
ortho(left, right, bottom, top, near, far)
Coord left, right, bottom, top, near, far;

subroutine ortho(left, right, bottom, top, near, far)
real left, right, bottom, top, near, far

procedure ortho(left, right, bottom, top, near, far: Coord);
```

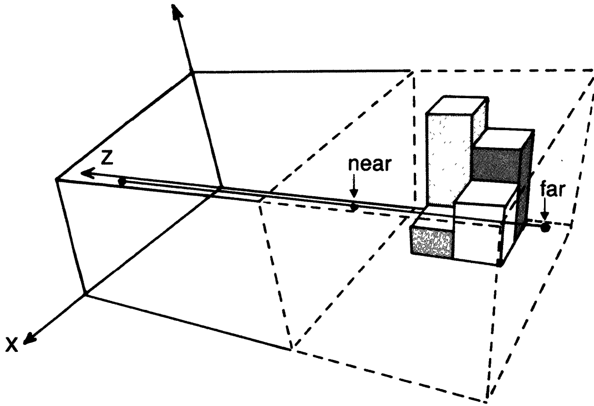
ortho2

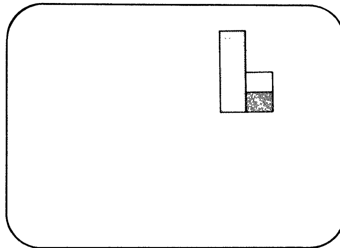
`ortho2` defines a 2-D clipping rectangle. When you use `ortho2` with 3-D world coordinates, the *z* values do not change.

```
ortho2(left, right, bottom, top)
Coord left, right, bottom, top;

subroutine ortho2(left, right, bottom, top)
real left, right, bottom, top

procedure ortho2(left, right, bottom, top: Coord);
```

4.4 Viewports

The *viewport* is the area of the window that displays an image. You specify it in window coordinates. The total visible screen area is 1024 pixels wide and 768 pixels high.





ortho (-5.,5.,-3.,3.,1.,3.);
translate (0.,0.,-2.);

ortho defines a viewing window in the x-y plane, looking down the -z axis. An orthographic view of the object between the near and far planes is projected onto the window.

Figure 4-7. ortho

viewport

`viewport` specifies, in window coordinates, the area of the window that displays an image. Specifying the viewport is the first thing you do when you map world coordinates to screen coordinates. Its arguments (*left*, *right*, *bottom*, *top*) define a rectangular area on the window by specifying the left, right, bottom, and top coordinates. The projection of the portion of world space that `window`, `ortho`, or `perspective` describe is mapped into the viewport.

```
viewport(left, right, bottom, top)
Screencoord left, right, bottom, top;

subroutine viewpo(left, right, bottom, top)
integer*4 left, right, bottom, top

procedure viewport(left, right, bottom, top: Screencoord);
```

getviewport

`getviewport` returns the current viewport. Its arguments (*left*, *right*, *bottom*, *top*) are the addresses of four memory locations. These are assigned the left, right, bottom, and top coordinates of the current viewport.

```
getviewport(left, right, bottom, top)
Screencoord *left, *right, *bottom, *top;

subroutine getvie(left, right, bottom, top)
integer*2 left, right, bottom, top

procedure getviewport(var left, right, bottom, top:
     Screencoord);
```

`viewport` sets both the viewport and the screenmask to the same area. (The *screenmask* is a specified rectangular area of the screen to which all drawings are clipped.) The viewport maps coordinates to the screen and the screenmask specifies the portion of the screen to which the geometry can be drawn. The screenmask is a setting that regards only the physical display within the window. The screenmask and viewport are usually set to the same area.

scrmask

`scrmask` sets only the screenmask, which should be placed entirely within the viewport. When the viewport is larger than the screenmask, character strings that begin inside the viewport are clipped to the screenmask. This process is called *fine clipping*. Character strings that begin outside the viewport are clipped out; this is called *gross clipping*. Figure 4-8 illustrates character clipping.

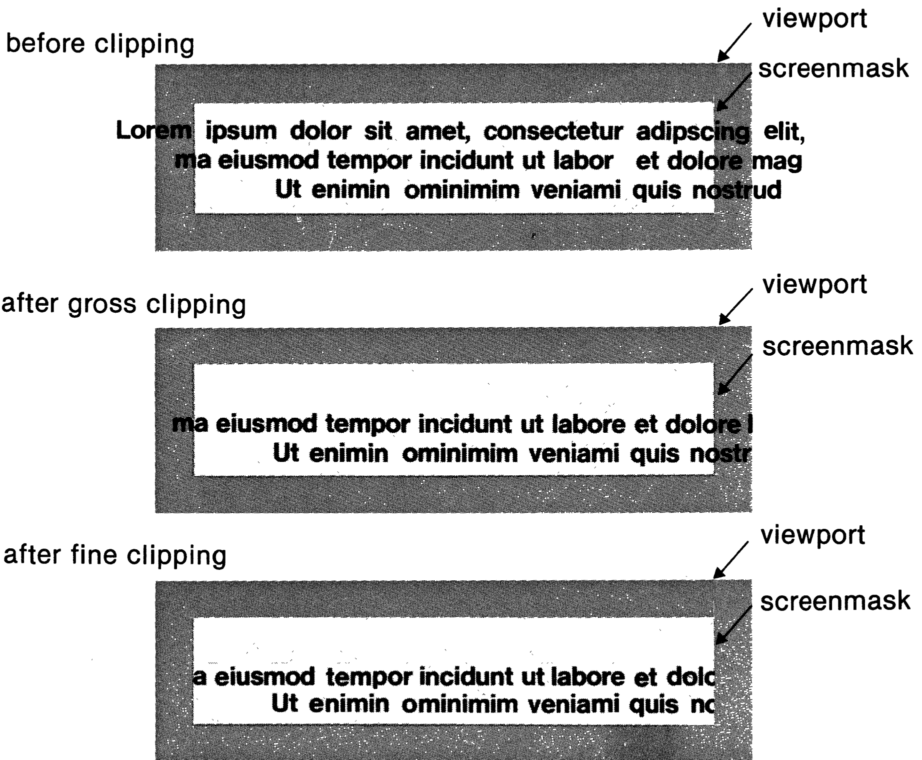
In addition to character strings, the system clips all other drawing primitives to the screenmask. This clipping occurs on a per-pixel basis after rasterization. Therefore, use gross clipping to a viewport to minimize the area of rasterization and use fine clipping to a screenmask for character strings.

```
scrmask(left, right, bottom, top)
Screencoord left, right, bottom, top;

subroutine scrmas(left, right, bottom, top)
integer*4 left, right, bottom, top

procedure scrmask(left, right, bottom, top:
     Screencoord);
```

getscrmask

`getscrmask` returns the coordinates of the current screenmask in the arguments *left*, *right*, *bottom*, and *top*.

```
getscrmask(left, right, bottom, top)
Screencoord *left, *right, *bottom, *top;

subroutine getscr(left, right, bottom, top)
integer*2 left, right, bottom, top

procedure getscrmask(var left, right, bottom, top:
     Screencoord);
```



Gross clipping removes all strings that start outside the viewport. Fine clipping trims individual characters to the screenmask.

Figure 4-8. Gross and Fine Clipping

pushviewport

The IRIS maintains a stack of viewports and the top element in the stack is the current viewport. `pushviewport` duplicates the current viewport and pushes it on the stack.

```
pushviewport()

subroutine pushvi

procedure pushviewport;
```

popviewport

`popviewport` pops the stack of viewports and sets the screenmask. The viewport on top of the stack is lost.

```
popviewport()

subroutine popvie

procedure popviewport;
```

4.5 User-Defined Transformations

A transformation changes the size and orientation of an object by modifying either the object itself or the position of the viewport. A transformation is expressed as a 4x4 floating point matrix. You can build complex transformations by linking a series of primitive transformation routines, such as `rotate`, `rot`, `translate`, or `scale`. If M, V, and P are modeling, viewing, and projection transformations, you can formulate transformation S, which maps object space into screen space, as follows:

$$S = M\ V\ P$$

$$[\,x\ y\ z\ w\,]\ M\ V\ P = [x'\ y'\ z'\ w']$$

The clipping boundaries are

$x = \pm w$, $y = \pm w$, and $z = \pm w$.

The resulting screen coordinates,

$$\frac{x}{w}, \frac{y}{w}, \text{ and } \frac{z}{w},$$

are scaled to the current viewport.

The Geometry Pipeline maintains a stack that holds up to 32 transformation matrices. (Eight of the the 32 matrices are in hardware.) The system applies the matrix on top of the stack, i.e., the *current transformation matrix,* to all coordinate data.

The Geometry Pipeline forms a complex transformation matrix by *premultiplying* the current matrix by each primitive transformation. It forms transformation S, defined above, by executing coordinate transformation routines in reverse order: first, projection routines; second, viewing routines; and third, modeling routines. Note that the Geometry Pipeline loads P onto the matrix stack, while both V and M premultiply the current matrix.

The projection, viewing, and modeling routines above provide a high-level interface that manages the transformation matrix stack. Additional routines allow direct control over the stack. These routines load or multiply user-defined transformation matrices, push and pop the transformation stack, and retrieve the matrix on the top of the stack.

loadmatrix

`loadmatrix` loads a 4x4 floating point matrix onto the stack, replacing the current top of the stack.

```
loadmatrix(m)
Matrix m;

subroutine loadma(m)
real m(4,4)

procedure loadmatrix(var m: Matrix);
```

multmatrix

`multmatrix` premultiplies the current top of the transformation stack by the given matrix; i.e., if *T* is the current matrix, `multmatrix(M)` replaces *T* with *MT*.

```
multmatrix(m)
Matrix m;

subroutine multma(m)
real m(4,4)

procedure multmatrix(var m: Matrix);
```

pushmatrix

`pushmatrix` pushes down the transformation stack, duplicating the current matrix. If the transformation stack contains one matrix, *M*, after a call to `pushmatrix`, it will contain two copies of *M*. You can modify only the top copy. For more information, see the *IRIS Programming Tutorial*, Chapter 7.

```
pushmatrix()

subroutine pushma

procedure pushmatrix;
```

popmatrix

`popmatrix` pops the transformation stack.

```
popmatrix()

subroutine popmat

procedure popmatrix;
```

getmatrix

`getmatrix` copies the transformation matrix from the top of the transformation stack to an array provided by the user; the stack does not change.

```
getmatrix(m)
Matrix m;
```

```
subroutine getmat(m)
real m(4,4)
```

```
procedure getmatrix(var m: Matrix);
```

5. Linestyles, Patterns, and Fonts

This chapter discusses routines that determine the characteristics, or *attributes*, of images the IRIS displays on the screen. Attributes include the values for:

- linestyle, which determines whether a line appears solid or as a series of dashes
- pattern, which determines the pattern with which shapes are filled
- font, which determines the font in which text strings appear

All of the above attributes determine precisely which pixels the IRIS draws when a drawing routine executes.

5.1 Linestyles

Linestyle is a 16-bit pattern the IRIS uses to draw lines on the monitor. The system runs this pattern repeatedly to determine which pixels in a 16-pixel line segment it must color. For example, the linestyle 0xFFFF draws a solid line; 0xFOFO draws a dashed line; and 0x8888 draws a dotted line. The least significant bit of the pattern is the mask for the first pixel of the line and every sixteenth pixel thereafter. There is no performance penalty for drawing lines that are not solid.

deflinestyle

`deflinestyle` defines a linestyle. Its arguments specify an index into a table *(n)*, which stores linestyles and a 16-bit linestyle pattern *(ls)*. There are 2^{16} possible linestyle patterns; you can define up to 65,535 of those patterns at one time. By default, index 0 contains linestyle 0xFFFF, which draws solid lines. You cannot redefine the linestyle at index 0.

If you redefine a linestyle, the previous linestyle definition is lost.

```
deflinestyle(n, ls)
short n;
Linestyle ls;

subroutine deflin(n, ls)
integer*4 n, ls

procedure deflinestyle(n: longint; ls: Linestyle);
```

setlinestyle

There is always a current linestyle; the IRIS uses it to draw lines and to outline rectangles, polygons, circles, and arcs. Linestyle 0 is the default linestyle. Use `setlinestyle` to select another linestyle. Its argument, *index*, is an index into the linestyle table built with calls to `deflinestyle`.

```
setlinestyle(index)
short index;

subroutine setlin(index)
integer*4 index

procedure setlinestyle(index: longint);
```

5.1.1 Modifying the Linestyle Pattern

Four routines modify the application of the linestyle pattern: `lsbackup`, `lsrepeat`, `resetls`, and `linewidth`. You can get the current values for these attributes using `getstyle`, `getlsbckup`, `getresetls`, `getlsrepeat`, and `getlwidth`.

`lsbackup` guarantees that a line has a clearly marked endpoint. Normally, the current linestyle is a rotating pattern:

```
for each pixel in the line {
        if low-order bit of pattern = 1 {
                write current color into pixel
        }
        rotate pattern right one bit;
        compute next pixel;
}
```

This algorithm implies that the line can end without a clearly marked endpoint.

lsbackup

`lsbackup` guarantees the last two pixels in a line are drawn, when enabled. It takes one Boolean argument. TRUE(1) enables backup mode. FALSE(0), the default setting, uses the linestyle as is and allows the line to have invisible endpoints.

```
lsbackup(b)
Boolean b;

subroutine lsback(b)
logical b

procedure lsbackup(b: longint);
```

resetls

The IRIS normally uses a fresh copy of the linestyle for each new line; this can affect the smooth appearance of curved shapes. `resetls(0)` allows the drawing of a series of line segments with a continuous pattern; this process uses many short, straight lines to approximate curved lines. It is useful for drawing circles, arcs, or curves. If you do not reset the linestyle between segments, the pattern of the curve appears smooth and continuous.

`resetls` has one Boolean argument. FALSE(0) turns off the mode, and the linestyle is not reset between segments. TRUE(1), the default, starts each line with a fresh copy of the pattern. `resetls` initializes the linestyle, no matter what argument is specified. If `resetls` is FALSE(0) when `setlinestyle` is called, the linestyle does not change until `resetls` is called. Set `resetls` to TRUE(1) when linestyle backup mode is enabled.

```
resetls(b)
Boolean b;

subroutine resetl(b)
logical b

procedure resetls(b: longint);
```

lsrepeat

`lsrepeat` creates linestyles that are longer than 16 bits. It multiplies each bit in the pattern by *factor*. Consequently, each 0 in the linestyle pattern becomes a series of *factor* x 0, and each 1 becomes a series of *factor* x 1. For example, if the line pattern is 0000000001111111 and *factor*=1, the linestyle is 9 bits off followed by 7 bits on. If *factor* =3, the linestyle is 27 bits off followed by 21 bits on.

```
lsrepeat(factor)
long factor;

subroutine lsrepe(factor)
integer*4 factor

procedure lsrepeat(factor: longint);
```

linewidth

`linewidth` specifies the width of a line. The IRIS measures the width in pixels along the x axis or along the y axis. It defines the width of a line as the number of pixels along the axis having the smallest difference between the endpoints of the line. If `linewidth` is set to $n > 1$, `resetls` must be TRUE(1) for reasonable results.

```
linewidth(n)
short n;

subroutine linewi(n)
integer*4 n

procedure linewidth(n: longint);
```

You can access the current values of the line drawing attributes with `getlstyle`, `getlsbackup`, `getresetls`, `getlsrepeat`, and `getlwidth`.

getlstyle

`getlstyle` returns the index of the current linestyle.

```
long getlstyle()

integer*4 function getlst()

function getlstyle: longint;
```

getlsbackup

`getlsbackup` returns the current value of the linestyle backup flag. TRUE(1) indicates that the last two pixels of a line are colored, regardless of the linestyle. FALSE(0), the default, indicates that the line can have invisible end points.

```
long getlsbackup()

logical function getlsb()

function getlsbackup: longint;
```

getresetls

`getresetls` returns the current value of the reset linestyle flag. TRUE(1), the default, indicates the system reinitializes the linestyle for each line segment. FALSE(0) indicates that the linestyle is continuous across line segment boundaries.

```
long getresetls()

logical function getres()

function getresetls: longint;
```

getlsrepeat

`getlsrepeat` returns the factor (integer) by which the linestyle is multiplied for patterns that are longer than 16 bits.

```
long getlsrepeat()

integer*4 function getlsr()

function getlsrepeat: longint;
```

getlwidth

`getlwidth` returns the current linewidth in pixels.

```
long getlwidth()

integer*4 function getlwi()

function getlwidth: longint;
```

5.2 Patterns

You can fill rectangles, polygons, and arcs with arbitrary patterns. A pattern is an array of short integers that defines a rectangular pixel array. The pattern controls which pixels the IRIS colors when it draws filled objects. The system aligns the pattern to the lower-left corner of the screen, rather than to the filled shape, so that it appears continuous over large areas.

defpattern

`defpattern` defines patterns. Its arguments specify an index into a table of patterns *(n)*, a size *(size)*, and an array of short integers *(mask)*. A pattern can be 16x16, 32x32, or 64x64. The origin of the pattern is the lower-left corner of the screen. You define the bottom row first. You specify each row of the pattern as a series of short integers for a 16x16 pattern. Figure 5-1 shows some possible patterns and their definitions in C. Pattern 0 is the default solid pattern, which you cannot change.

```
defpattern(n, size, mask)
short n, size;
short *mask;

subroutine defpat(n, size, mask)
integer*4 n, size
integer*2 mask((size*size)/16)

procedure defpattern(n: size: longint; var mask: Short);
```

setpattern

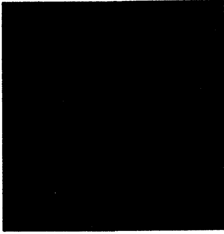
`setpattern` selects a defined pattern that the IRIS uses. `defpattern` provides an index that you use as the argument for `setpattern`. Pattern 0 is the default solid pattern. Shading works only with the solid pattern.

```
setpattern(index)
short index;

subroutine setpat(index)
integer*4 index

procedure setpattern(index: longint);
```

getpattern

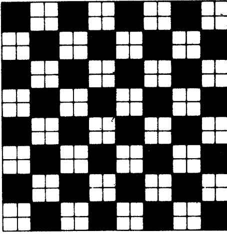
`getpattern` returns the index of the current pattern.

```
long getpattern()

integer*4 function getpat()

function getpattern: longint;
```



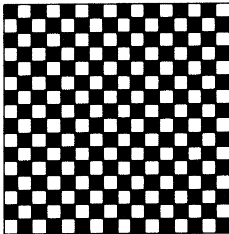
Pattern solid

= { 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF }



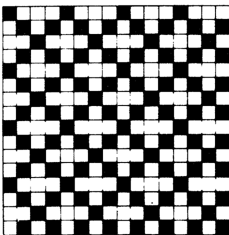
Pattern checked

= { 0x3333, 0x3333, 0xCCCC, 0xCCCC, 0x3333, 0x3333, 0xCCCC, 0xCCCC,
0x3333, 0x3333, 0xCCCC, 0xCCCC, 0x3333, 0x3333, 0xCCCC, 0xCCCC }



Pattern halftone

= { 0x5555, 0xAAAA, 0x5555, 0xAAAA, 0x5555, 0xAAAA, 0x5555, 0xAAAA,
0x5555, 0xAAAA, 0x5555, 0xAAAA, 0x5555, 0xAAAA, 0x5555, 0xAAAA }



Pattern crosshatch

= { 0x5555, 0x2222, 0x5555, 0x8888, 0x5555, 0x2222, 0x5555, 0x8888,
0x5555, 0x2222, 0x5555, 0x8888, 0x5555, 0x2222, 0x5555, 0x8888 }

A pattern is a 16X16, 32X32, or 64X64 array of bits with the origin in the lower-left corner.

Figure 5-1. Sample Patterns

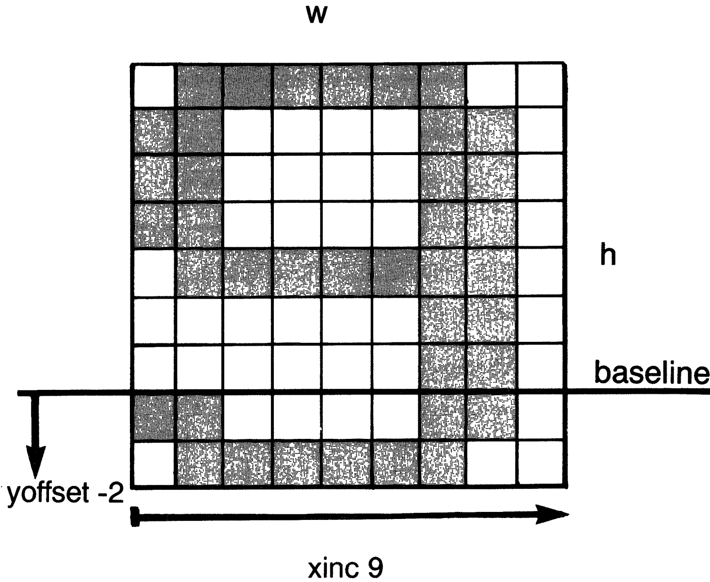
5.3 Fonts

defrasterfont

`defrasterfont` defines a raster font. It has six arguments. *n* is an index into the font table; *ht* is an integer that specifies the maximum height of the font characters in pixels. *nc* gives the number of characters in the font, which is the number of elements in the *chars* array. *chars* contains a description of each character in the font. The description includes the height and width of the character in pixels; the offsets from the character origin to the lower-left corner of the bounding box; an offset into the array of rasters, and the amount to add to the current character *x* position after drawing the character. Figure 5-2 gives a sample character definition. *raster* is an array of *nr* shorts of bitmap information. It is a one-dimensional array of mask bytes ordered from left to right, then bottom to top. Mask bits are left-justified in the character's bounding box.

```
defrasterfont(n, ht, nc, chars, nr, raster)
short n, ht, nc, nr;
Fontchar chars[];
short raster[];

subroutine defras(n, ht, nc, chars, nr, raster)
integer*4 n, ht, nc, nr
integer*2 raster(nr), chars(4*nc)

procedure defrasterfont(n, ht, nr, nc: longint; var nr:
     Fontchr; chars: longint; var raster: Short);
```

Font 0 is the default raster font, which you cannot redefine. It is a Helvetica-like font with fixed-pitch characters. If the viewport is set to the whole screen, approximately 110 of the default characters fit on a line (1 character occupies 9 pixels). If baselines are 16 pixels apart, 48 lines fit on the screen.




```
defrasterfont (n, ht, nc, chars, nr, rasters);

chars ['g']  = {    724,          8,    9,     0,       −2,      9 }
                byte offset      w     h   xoffset  yoffset  xinc
                into rasterarray
short
rasterarray [ ]     = { . . .
                        . . .
position 724 >          0x7E00,  0xC300,  0x0300,  0x0300,
                        0x7F00,  0xC300,  0xC300,  0xC300,
                        0x7E00,
                        . . .
                      }
```

Raster font characters are defined by a bitmap, 1 bit per pixel. The width and height of the character, the number of bits in one row of the bitmap, and the baseline position are also specified. See the manual page *defrasterfont* in the Reference Guide for a more complete example.

Figure 5-2. Sample Character Definition

font

`font` selects the font the IRIS uses whenever `charstr` draws a text string. Its argument is an index into the font table *(fntnum)* built by `defrasterfont`. This font remains the current font until another `font` executes.

```
font(fntnum)
short fntnum;

subroutine font(fntnum)
integer*4 fntnum

procedure font(fntnum: longint);
```

getfont

`getfont` returns the index of the current raster font.

```
long getfont()

integer*4 function getfon()

function getfont: longint;
```

getheight

`getheight` returns the maximum height of a character in the current raster font, including ascenders (present in tall characters, such as the letters t and h) and descenders (present in such characters as the letters y and p, which descend below the baseline). It returns the height in pixels.

```
long getheight()

integer*4 function gethei()

function getheight: longint;
```

getdescender

`getdescender` returns the longest descender in the current font. It returns the number of pixels the longest descender goes below the baseline.

```
long getdescender():

integer*4 function getdes()

function getdescender: longint;
```

strwidth

`strwidth` returns the width of a text string in pixels, using the character spacing parameters in the current raster font.

```
long strwidth(str)
String str;

integer*4 function strwid(str, length)
character*(*) str
integer*4 length

function strwidth(str: pstring128): longint;
```

6. Display and Color Modes

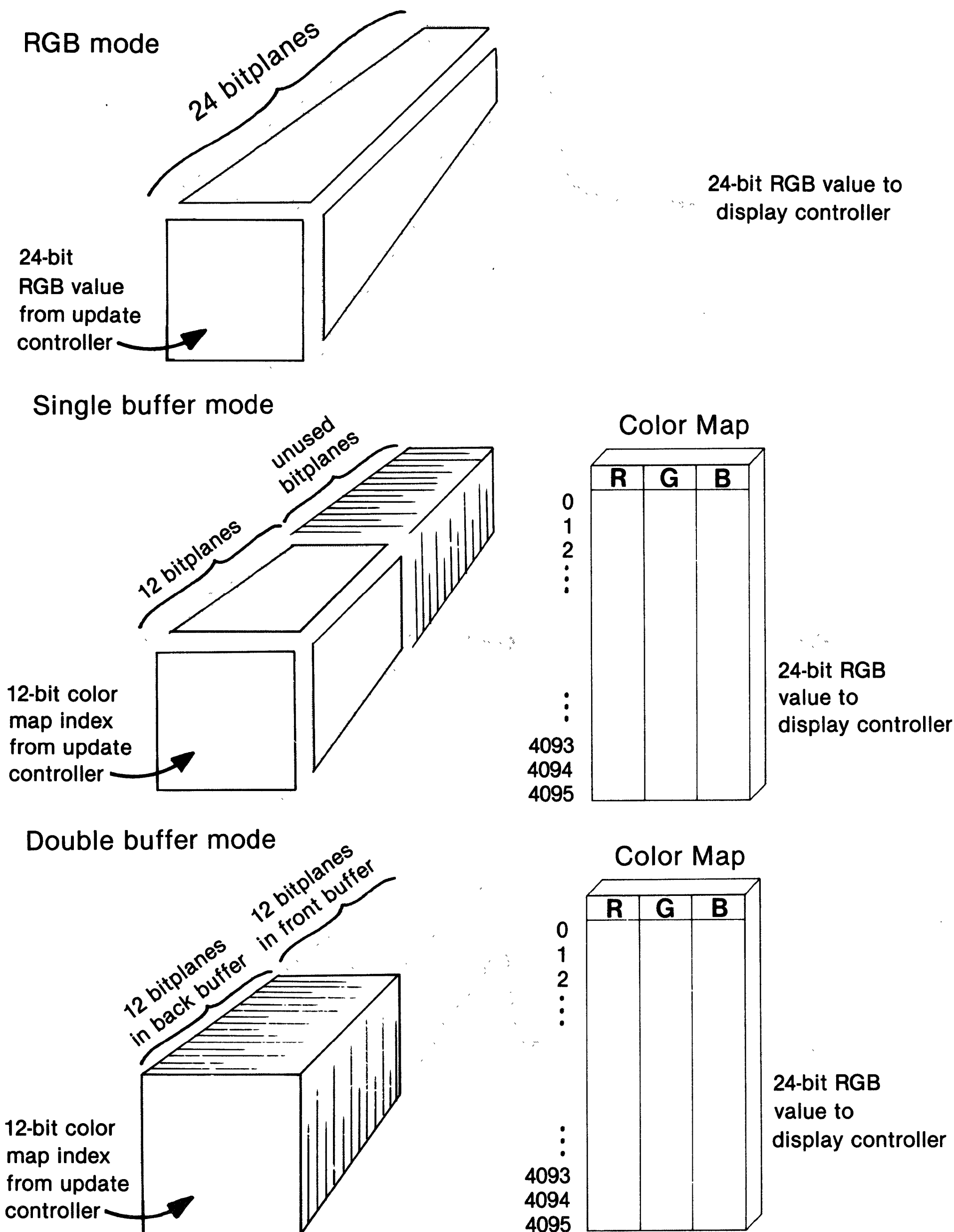
Display and color modes tell the IRIS how to color the pixels. When the IRIS draws a pixel, it writes a color value for it into the bitplanes. Each bitplane corresponds to one of the bits that hold the color index value for the pixel. For example, if your IRIS has 12 bitplanes, 12 bits are devoted to each pixel. The bitplanes store the screen image the drawing routines create.

The IRIS offers three display modes (single buffer, double buffer, and RGB), which determine how it stores the color values in the bitplanes and how the system uses the values to display a screen image. A fourth mode, z-buffer, which removes hidden surfaces is discussed in Chapter 12, Hidden Surfaces. For a more elementary discussion of the display and color modes, see the *IRIS Programming Tutorial*, Chapter 4.

6.1 Display Modes

The IRIS stores the screen image in a set of 4 to 24 bitplanes (Figure 6-1). Each bitplane provides 1 bit of storage per pixel. The corresponding locations in all bitplanes represent values (from 4 to 24 bits) for each pixel. These values determine the color of the pixel when it is displayed on the screen. The three *display modes*—single buffer, double buffer, and RGB—specify how the IRIS stores values in the bitplanes and how it uses them.

The display mode routines, `RGBmode`, `singlebuffer`, and `doublebuffer` take effect only after a call to `gconfig`.



In RGB mode, 24-bit RGB values are stored in the bitplanes. In single buffer and double buffer modes, the bitplanes store 12-bit color map indices.

Figure 6-1. Bitplanes and Color Maps

gconfig

`gconfig` defines the mapping from colors to bitplanes using the display mode, color map mode (discussed in Section 6.2), and the number of available bitplanes.

```
gconfig()

subroutine gconfi

procedure gconfig;
```

6.1.1 RGB Mode

The RGB value determines the color of a pixel. An RGB value consists of three 8-bit intensity values—one for red, one for green, and one for blue. The IRIS loads these values into the three digital-to-analog converters, which control the color and brightness of each pixel.

RGBmode

`RGBmode` puts the system into RGB mode. In *RGB mode*, the IRIS writes an RGB value into the bitplanes when it draws a pixel. Use RGB mode only when the system has at least 24 bitplanes (room for three 8-bit values).

RGBmode takes effect after a call to `gconfig`.

Note: Do not use RGB mode while running the window manager.

```
RGBmode()

subroutine RGBmod

procedure RGBmode;
```

6.1.2 Single Buffer and Double Buffer Modes

Single buffer and *double buffer modes* provide other ways to use the bitplanes. In these modes, the values contained in the bitplanes are indices into a color map. The color map is a table of RGB values. This means that a system with fewer than 24 bitplanes can still display the full range of colors specified by 24-bit RGB values. For example, although each pixel in

a system with 4 bitplanes can store only 16 different values, the colors specified in the color map can be any of the 2^{24}=16.8 million possible RGB colors. You can divide the color map into 16 smaller maps. (See Section 6.2 for further discussion of color maps.)

singlebuffer

In single buffer mode, the IRIS uses up to 12 bitplanes to store the color map indices. It does not use any extra bitplanes or uses them for z-buffering if required (see Chapter 12, Hidden Surfaces). Single buffer mode simultaneously updates and displays images, so incomplete or changing pictures can appear on the screen. `singlebuffer` invokes this display mode. In single buffer mode, the window manager requires 2 bitplanes, leaving 10 available for user applications.

`singlebuffer` takes effect after a call to `gconfig`.

```
singlebuffer()

subroutine single

procedure singlebuffer;
```

doublebuffer

Double buffer mode separates the bitplanes into two groups, called *front buffer* and *back buffer*. By default, the IRIS displays the front buffer, while the drawing routines update the back buffer. This allows the system to display a complete image while it draws a new one. In double buffer mode, the window manager requires 4 bitplanes, leaving 10 available for user applications.

`doublebuffer` takes effect after a call to `gconfig`.

```
doublebuffer()

subroutine double

procedure doublebuffer;
```

swapbuffers

`swapbuffers` swaps the front and back buffers during the vertical retrace period. After the IRIS draws an image in the back buffer, `swapbuffers` displays it.

```
swapbuffers()
```

```
subroutine swapbu
```

```
procedure swapbuffers;
```

swapinterval

`swapinterval` establishes a minimum time between buffer swaps. If you specify a swap interval of 5, the screen is refreshed at least five times between execution of successive calls to `swapinterval`. `swapinterval` provides a way to change frames at a steady rate if the system can create a new image within one swap interval. The default interval is 1. `swapinterval` is valid only in double buffer mode; it is ignored in single buffer and RGB modes. `swapinterval` takes effect after the next call to `swapbuffers`.

```
swapinterval(i)
short i;
```

```
subroutine swapin(i)
integer*4 i
```

```
procedure swapinterval(i: Short);
```

backbuffer

It is sometimes convenient to update both the front and the back buffers, or to update the front buffer instead of the back one. `backbuffer` enables updating in the back buffer. Its argument is a Boolean value. The backbuffer is enabled when *b* is TRUE(1), which is the default value. When *b* is FALSE(0), the back buffer is not enabled for writing.

```
backbuffer(b)
Boolean b;
```

```
subroutine backbu(b)
logical b
```

```
procedure backbuffer(b: longint);
```

frontbuffer

`frontbuffer` enables updating of the front buffer. Its argument is a Boolean value. The front buffer is not enabled when *b* is FALSE(0), which is the default value. When *b* is TRUE(1), the front buffer is enabled for writing.

```
frontbuffer(b)
Boolean b;

subroutine frontb(b)
logical b

procedure frontbuffer(b: longint);
```

getbuffer

`getbuffer` indicates which buffer(s) is enabled for writing. 1, the default value, means the back buffer is enabled; 2 means that the front buffer is enabled; and 3 means that both are enabled. `getbuffer` returns 0 if neither buffer is enabled or if the IRIS is not in double buffer mode.

```
long getbuffer()

integer*4 function getbuf()

function getbuffer: longint;
```

getdisplaymode

`getdisplaymode` returns the current display mode. 0 indicates RGB mode; 1 indicates single buffer mode; and 2 indicates double buffer mode.

```
long getdisplaymode()

integer*4 function getdis()

function getdisplaymode: longint;
```

getplanes

getplanes returns the number of available bitplanes. For example, a 24-bitplane system returns 24 available bitplanes in RGB mode and 12 available bitplanes in single buffer and double buffer modes. An 8-bitplane system returns 8 available bitplanes in single buffer mode and 4 available bitplanes in double buffer mode. getplanes cannot be used in RGB mode with fewer than 24 bitplanes.

```
long getplanes()
```

```
integer*4 function getpla()
```

```
function getplanes: longint;
```

gsync

In single buffer and RGB modes, rapidly changing scenes should be synchronized with the screen refresh rate. gsync waits for the next vertical retrace period.

```
gsync()
```

```
subroutine gsync
```

```
procedure gsync;
```

finish

You use finish only on an IRIS terminal that is running remote graphics. It is useful when there are network and pipeline delays. finish blocks the host process until all prior routines execute. It forces all unsent routines down the network/graphics pipeline to the bitplanes; sends a final token; and blocks until that token has gone through the network and graphics pipeline, and the remote graphics terminal acknowledges completion.

```
finish()
```

```
subroutine finish
```

```
procedure finish;
```

6.2 Color Maps

RGB mode writes 8 bits each of red, green, and blue intensities into the bitplanes. It offers a palette of 2^{24}=16.8 million different colors. A disadvantage of RGB mode is that it is only effective in a system with at least 24 bitplanes. *Color mapping* is a flexible technique—it stores indices into the color map in the bitplanes, instead of RGB values. The color map is a table of 24-bit RGB values. It stores 2^{12}=4096 RGB values. Consequently, a system with 8 bitplanes addresses only 2^{8}=256 of the 4096 entries in the color map. Each of those entries has the full 24-bit precision of RGB mode.

mapcolor

`mapcolor` sets a color map entry to a specified RGB value. Its arguments are a color map index *(color)* and 8 bits each of *red, green*, and *blue* intensities. Pixels written with the color specified by *color* are displayed with the specified RGB intensities. See Section 6.3 for a discussion of the `color` routines. In multimap mode, `mapcolor` updates only the current color map. It ignores invalid indices.

```
mapcolor(color, red, green, blue)
Colorindex color;
short red, green, blue;

subroutine mapcol(color, red, green, blue)
integer*4 color, red, green, blue
integer*2 r, g, b,

procedure mapcolor(color: longint; red, green,
    blue: longint);
```

getmcolor

`getmcolor` returns the red, green, and blue components of a color map entry.

```
getmcolor(color, red, green, blue)
Colorindex color;
short*red, *green, *blue;

subroutine getmco(color, red, green, blue)
integer*4 color, red, green, blue

procedure getmcolor(color: longint; var red, green, blue:
    Short);
```

You can use the color map in either `onemap` mode, the default, which organizes the color map as described above—a single map with room for 4096 RGB entries; or `multimap` mode, which organizes the color map as 16 independent maps, each with a maximum of 256 RGB entries.

onemap

`onemap` organizes the color map as a single map with a maximum of 4096 RGB entries. The number of entries is 2^p where p is the number of available bitplanes.

`onemap` takes effect after a call to `gconfig`.

Note: Color map indices are limited to 12 bits in onemap mode.

```
onemap()

subroutine onemap

procedure onemap;
```

multimap

`multimap` organizes the color map as 16 small maps. There are two advantages to multimap mode:

- It allows you to rapidly switch among 16 different maps, each of which defines up to 256 different colors.
- It provides an additional tool for altering screen images (e.g., an image can be color-inverted by switching to a different color map).

`multimap` takes effect after a call to `gconfig` (see Section 6.1).

Note: Color map indices are limited to 8 bits in multimap mode.

```
multimap()
```

```
subroutine multim
```

```
procedure multimap;
```

getcmmode

`getcmmode` returns the current color map mode. 0 indicates multimap mode; 1 indicates onemap mode.

```
long getcmmode()
```

```
integer*4 function getcmm()
```

```
function getcmmode: longint;
```

setmap

`setmap` selects which of the small maps (0 through 15) the IRIS uses in multimap mode.

```
setmap(mapnum)
short mapnum;
```

```
subroutine setmap(mapnum)
integer*4 mapnum
```

```
procedure setmap(mapnum: Scoord);
```

getmap

`getmap` returns the number (from 0 to 15) of the current color map. 0 indicates onemap mode.

```
long getmap()
```

```
integer*4 function getmap()
```

```
function getmap: longint;
```

cyclemap

`cyclemap` cycles through color maps at a specified rate. It defines a duration (in vertical retraces), the current map, and the map that follows when the duration lapses. For example, the following routines set up multimap mode and cycle between two maps, leaving map 1 on for ten vertical retraces and map 3 on for five retraces.

```
multimap();
gconfig();
cyclemap(10, 1, 3);
cyclemap(5, 3, 1);
```

Note: Before you exit your program, call `cyclemap` with all durations set to zero. `cyclemap` settings remain in effect after you exit a program.

```
cyclemap(duration, map, nextmap)
short duration, map, nextmap;

subroutine cyclem(duration, map, nextmap)
integer*4 duration, map, nextmap

procedure cyclemap(duration, map, nextmap: Short);
```

blink

`blink` changes the color map entry at a specified rate. It specifies a blink rate *(rate)*, a color map index *(color)*, and *red*, *green*, and *blue* values. *rate* indicates the number of vertical retraces at which the IRIS updates the color located at *color* in the current color map. *color*'s value is either the original value or the new value supplied by *red*, *green*, and *blue*. Up to 20 colors can blink simultaneously, each at a different rate. You can change the blink rate by calling `blink` a second time with the same *color* but a different *rate*. To terminate blinking and restore the original color, call `blink` with *rate* = 0 when `color` specifies a blinking color map entry.

Note: Program termination does not stop this routine; you must explicitly set all durations to zero.

```
blink(rate, color, red, green, blue)
short rate;
Colorindex color;
short red, green, blue;

subroutine blink(rate, color, red, green, blue)
integer*4 rate, color, red, green, blue

procedure blink(rate: longint; color: longint; red, green,
    blue: longint);
```

6.3 Colors and Writemasks

This section describes the routines that set the current color. When the IRIS draws a pixel in single buffer or double buffer mode, it writes the current color map index into the bitplanes.

6.3.1 Colors

color

`color` sets the current color index in single buffer and double buffer modes. In onemap mode, the index is in the range of 0 to 4095. These routines work differently in multimap mode; see Section 6.2 for more information. For more elementary information on color see the *IRIS Programming Tutorial*, Chapter 5.

```
color(c)
Colorindex c;

subroutine color(c)
integer*4 c

procedure color(c: longint);
```

The program below draws a blue rectangle around a red circle. Here red and blue are merely indices into the color map. They are defined in the file *gl.h* and correspond to the color map entries that `ginit` set for the colors red and blue.

■ C Program: BLUE RECTANGLE

```
#include "gl.h"

main()
{
    ginit();
    color(BLUE);
    recti(0, 0, 100, 100);
    color(RED);
    circi(50, 50, 50);
    gexit();
}
```

■ FORTRAN Program: BLUE RECTANGLE

```
    #INCLUDE /usr/include/fgl.h
    #INCLUDE /usr/include/fdevice.h

    CALL GINIT
    CALL COLOR(BLUE)
    CALL RECTI(0,0,100,100)
    CALL COLOR(RED)
    CALL CIRCI(50,50,50)
  5 continue
    if(.not. getbut(RIGHTM))go to 5
    call color(BLACK)
    call clear
    CALL GEXIT
    STOP
    END
```

getcolor

`getcolor` returns the current color. The system must be in single buffer or double buffer mode when it executes.

```
long getcolor()

integer*4 function getcol()

function getcolor: longint;
```

RGBcolor

`RGBcolor` sets the current color in RGB mode. The lower-order 8 bits of the three arguments are the intensity values for the colors red, green, and blue. The IRIS writes these numbers into the bitplanes whenever it draws a pixel; they directly control the intensity of the red, green, and blue the screen displays.

Note: Do not use RGB mode under the window manager.

```
RGBcolor(red, green, blue)
short red, green, blue;

subroutine RGBcol(red, green, blue)
integer*4 red, green, blue

procedure RGBcolor(red, green, blue: longint);
```

The following program draws a blue rectangle around a red circle in RGB mode.

■ C Program: BLUE RECTANGLE (In RGB Mode)

```
#include "gl.h"

main()
{
    ginit();
    RGBmode();
    gconfig();
    RGBcolor(0, 0, 225);
    recti(0, 0, 6, 6);
    RGBcolor(225, 0, 0);
    circi(3, 3, 2);
```

```
    gexit();
}
```

■ FORTRAN Program: BLUE RECTANGLE (in RGB Mode)

```
      #INCLUDE /usr/include/fgl.h
      #INCLUDE /usr/include/fdevice.h

      call ginit
      CALL RGBMOD
      CALL GCONFI
      CALL RGBCOL(0,0,225)
      CALL RECTI(0,0,6,6)
      CALL RGBCOL(225,0,0)
      CALL CIRCI(3,3,2)
 999    continue
      if(.not. getbut(RIGHTM))go to 999
      call greset
      call single
      call color(BLACK)
      call clear
      CALL GEXIT
      STOP
      END
```

6.3.2 Writemasks

To draw an object that moves across the screen while the background remains constant, you need to redraw both the background and the object each time the object moves to a new location. You can use writemasks to protect the bitplane values of the background and avoid redrawing the background. This keeps the background color constant, while the values of the moving object change.

Writemasks layer the images that appear on the screen. For example, you can write color values for the cursor into one bitplane. Using a writemask, you can shield this bitplane from ordinary drawing routines (which write values in the remaining bitplanes). As a result, the cursor always appears on the screen. See Figures 6-2 and 6-3.

writemask

In single buffer or double buffer mode, `writemask` determines which bitplanes the drawing routines affect. Its argument, *wtm*, is a mask with 1 bit per available bitplane. The bitplanes included in the writemask are enabled for writing. The corresponding bit in the current color index is written into the bitplane wherever a pixel is drawn. Zeros in the writemask mark bitplanes as read-only. These bitplanes do not change, regardless of the bits in the current color.

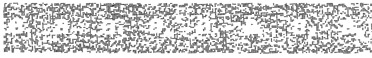
```
writemask(wtm)
Colorindex wtm;

subroutine writem(wtm)
integer*4 wtm

procedure writemask(wtm: longint);
```

getwritemask

`getwritemask` returns the current writemask. It is an integer with up to 12 significant bits, one for each available bitplane.

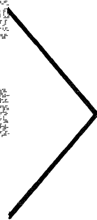
```
long getwritemask()

integer*4 function getwri()

function getwritemask: longint;
```

new color index



writemask

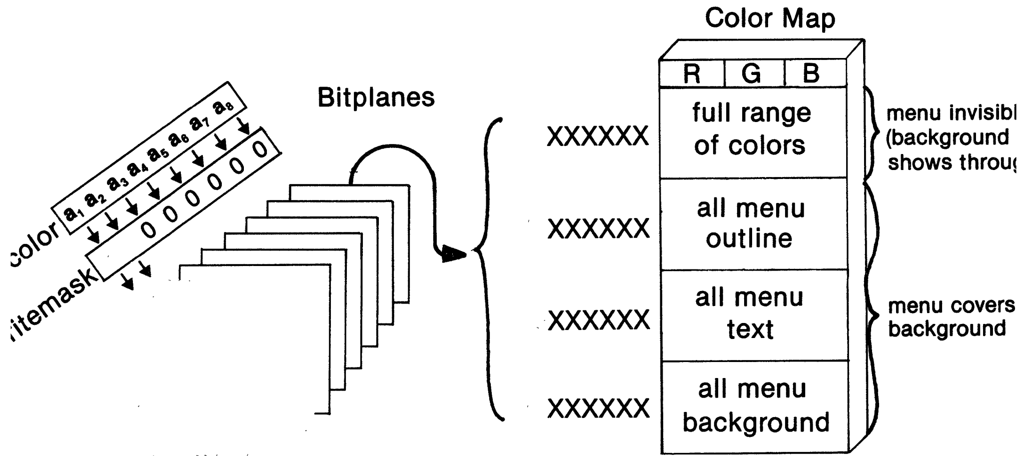
current color index in bitplanes

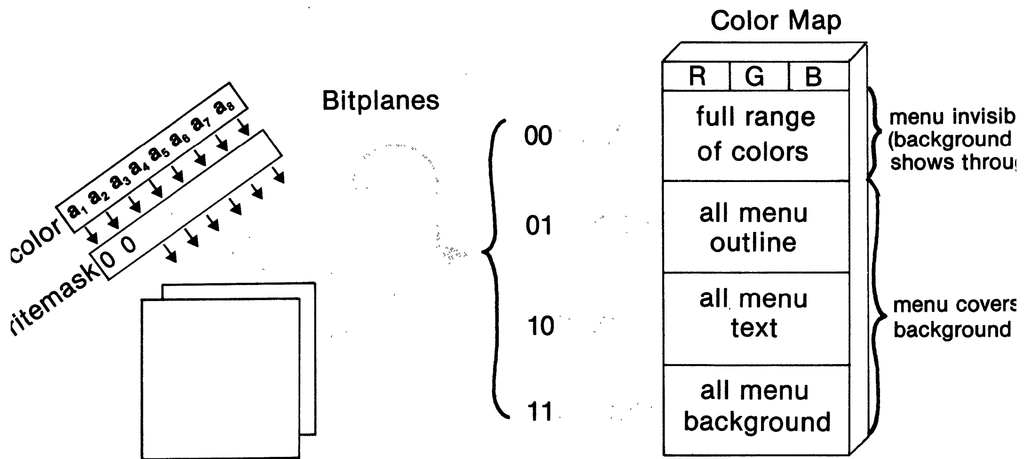


final color index

Writemasks determine whether or not a new value can be stored in each bitplane. A "1" in the writemask allows the system to store a new value (0 or 1) in the corresponding bitplane. A "0" prevents the system from storing a new value and the corresponding bitplane retains its current value. In this example, the values in the first and second bits (b_1 and b_2) do not change because the corresponding positions in the writemask are zero. All the other values (originally b_3,b_4, ... b_8) change to a_3,a_4, ... a_8 because the corresponding positions in the writemask are 1. Each value a_1, ... a_8 and b_1, ... b_8 is either 0 or 1.

Figure 6-2. Writemask





Writemasks create a layering of images by preserving the contents of specific bitplanes. In this example, the first two bitplanes are reserved for a pop-up menu. Since two bitplanes provide four possible values, the menu can use four different parts of the color map. If the menu is invisible (or not positioned) at a particular location, the remaining bitplanes determine the color value for that location.

Figure 6-3. Writemasks and the Color Map

RGBwritemask

In RGB mode, `RGBwritemask` masks bitplanes. Its three arguments, *red*, *green*, and *blue*, are masks for each of three sets of 8 bitplanes.

```
RGBwritemask(red, green, blue)
short red, green, blue;

subroutine RGBwri(red, green, blue)
integer*4 red, green, blue

procedure RGBwritemask(red, green, blue: longint);
```

gRGBmask

`gRGBmask` returns the current RGB writemask as three 8-bit masks. The masks are placed in the locations addressed by *redm*, *greenm*, and *bluem*.

```
gRGBmask(redm, greenm, bluem)
short *redm, *greenm, *bluem;

subroutine gRGBma(redm, greenm, bluem)
integer*2 redm, greenm, bluem

procedure gRGBmask(var redm, greenm, bluem: Short);
```

6.4 Cursors

A *cursor* is a 16x16 array of bits, which shows the current position of a graphics input device (see Chapter 7, Input/Output Routines).

defcursor

`defcursor` defines an entry in a table of cursors. As with linestyles and patterns, its arguments are a table index and a 16x16 bitmap. By default, cursor 0 is an arrow and cannot be overwritten. Figure 6-4 shows some examples of cursors and their definitions in C.

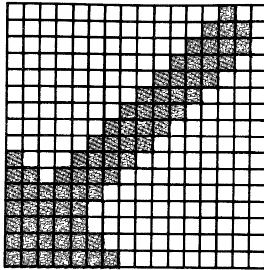
```
defcursor(n, curs)
short n;
Cursor curs;

subroutine defcur(n, curs)
integer*4 n
integer*2 curs(16)

procedure defcursor(n: longint; var curs: Cursor);
```

setcursor

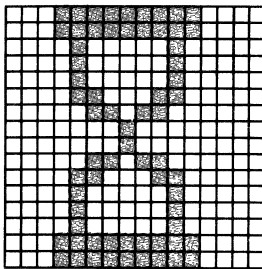
`setcursor` specifies the characteristics of the cursor. The first argument, *index*, picks a cursor from the definition table. *color* and *wtm* set a color map index and writemask for the cursor.

Note: The system ignores *color* and *wtm* under the window manager.

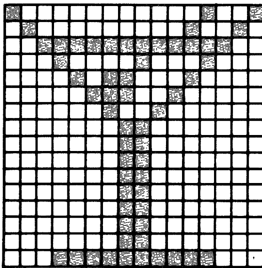
```
setcursor(index, color, wtm)
short index;
Colorindex color, wtm;

subroutine setcur(index, color, wtm)
integer*4 index, color, wtm

procedure setcursor(index: longint; color, wtm: longint);
```



Cursor arrow = { 0 x FE00, 0 x FC00, 0 x F800, 0 x F800,
0 x FC00, 0 x DE00, 0 x 8F00, 0 x 0780,
0 x 03C0, 0 x 01E0, 0 x 00F0, 0 x 0078,
0 x 003C, 0 x 001E, 0 x 000E, 0 x 0004}



Cursor hourglass = { 0 x 1FF0, 0 x 1FF0, 0 x 0820, 0 x 0820,
0 x 0820, 0 x 0C60, 0 x 06C0, 0 x 0100,
0 x 0100, 0 x 06C0, 0 x 0C60, 0 x 0820,
0 x 0820, 0 x 0820, 0 x 1FF0, 0 x 1FF0}



Cursor martini = { 0 x 1FF8, 0 x 0180, 0 x 0180, 0 x 0180,
0 x 0180, 0 x 0180, 0 x 0180, 0 x 0180,
0 x 0180, 0 x 0240, 0 x 0720, 0 x 0B10,
0 x 1088, 0 x 3FFC, 0 x 4022, 0 x 8011}

A cursor is a 16X16 array of bits with the origin in the lower-left corner. The cursor is defined bottom-up, just as raster characters are defined.

Figure 6-4. Sample Cursors

curorigin

`curorigin` sets the origin of a cursor (i.e., the place on the cursor that is aligned with the valuators that control the cursor position). *n* is the index of the cursor defined by `defcursor`. In RGB mode, use `RGBcursor` discussed below.

```
curorigin(n, xorigin, yorigin)
short n, xorigin, yorigin;

subroutine curori(n, xorigin, yorigin)
integer*4 n, xorigin, yorigin

procedure curorigin(n, xorigin, yorigin: longint);
```

getcursor

`getcursor` returns the index of the glyph, the color, and the writemask associated with the cursor and a Boolean value, which indicates whether the system automatically displays and updates the cursor. Its arguments are addresses of four locations where the four cursor attributes are to be returned. The default is the glyph at index 0 into the cursor table; it is displayed with the color 1 and drawn into all bitplanes. On each vertical retrace, the system automatically displays and updates the glyph. Use `RGBcursor` in RGB mode.

```
getcursor(index, color, wtm, b)
short *index;
Colorindex *color, *wtm;
Boolean *b;

subroutine getcur(index, color, wtm, b)
integer*2 index, color, wtm
logical b

procedure getcursor(var index: Short; var color, wtm:
    Colorindex; var b: Boolean);
```

RGBcursor

`RGBcursor` allows selection of a cursor from a table of user-defined 16x16-bit patterns. Its first argument, *index*, picks a glyph from the definition table. *red*, *green*, and *blue* specify the cursor color in RGB mode, while *redm*, *greenm*, and *bluem* define an RGB writemask for the cursor.

```
RGBcursor(index,red,green,blue,redm,greenm,bluem)
short index;
RGBvalue red,green,blue,redm,greenm,bluem;

subroutine RGBcur(index,red,green,blue,redm,greenm,bluem)
integer*2 index,red,green,blue,redm,greenm,bluem

procedure RGBcursor(index,red,green,blue,redm,greenm,bluem:
    longint);
```

gRGBcursor

`gRGBcursor` returns the six parameters of the `RGBcursor` and a Boolean value, which indicates whether the the system automatically updates the cursor position. The system must be in RGB mode when this routine executes.

```
gRGBcursor(index,red,green,blue,redm,greenm,bluem,b)
short*index,*red,*green,*blue,*redm,*greenm,*bluem;
Boolean*b;

subroutine gRGBcu(index,red,green,blue,redm,greenm,bluem,b)
integer*2 index,red,green,blue,redm,greenm,bluem
logical b

procedure gRGBcursor(var index,red,green,blue,redm,greenm,
    bluem: Short; var b: Boolean);
```

By default, the cursor is always displayed. As it is drawn on the screen, the system saves the image that it covers. When the cursor moves, it restores the saved image. In single buffer and RGB modes (and in double buffer mode when the front buffer is enabled), the image can change while the cursor is displayed so the saved image may no longer be valid. Accordingly, there are routines that turn the cursor on and off.

cursoff

`cursoff` removes the cursor from the screen. This routine should precede any drawing routines that can write into the cursor bitplanes.

```
cursoff()

subroutine cursof

procedure cursoff;
```

curson

`curson` tells the system to update and display a cursor automatically. `curson` is usually paired with `cursoff`, although you can call `curson` when the automatic cursor is visible.

```
curson()

subroutine curson

procedure curson;
```

The code segment below shows a typical drawing sequence in single buffer mode.

```
{
        greset();

        color(BLUE);
        cursoff();
        move2i(1, 0);
        draw2i(3, 6);
        draw2i(5, 0);
        move2i(4, 3);
        draw2i(2, 3);
        curson();
}
```

The following is a FORTRAN version of the above code segment with some modification. This code draws the character "A":

```
          #include /usr/include/fgl.h
          #include /usr/include/fdevice.h

          call ginit
          call color(BLACK)
          call cursof
          call clear
          call color(BLUE)
          call move2i(100,100)
          call draw2i(300,600)
          call draw2i(500,100)
          call move2i(410,325)
          call draw2i(190,325)
999       continue
          if (.not. getbut(RIGHTM))go to 999
          call color(BLACK)
          call clear
          call curson
          call gexit
          stop
          end
```

Whether or not the cursor is visible, the IRIS automatically updates the cursor position to reflect the values of the valuator devices to which it is attached (see Chapter 7, Section 7.1, for a discussion of `attachcursor`).

blankscreen

`blankscreen` turns the screen refresh on and off. Its argument is a Boolean value. TRUE(1) stops display; FALSE(0) restarts display. When bitplanes are simultaneously viewed and updated (as in single buffer and RGB modes, or when the front buffer is displayed in double buffer mode), there is competition for memory. This reduces performance, especially with noninterlaced monitors. To speed up drawing in these cases, turn off the display while drawing.

```
blankscreen(bool)
Boolean bool;

subroutine blanks(bool)
logical bool

procedure blankscreen(bool: Boolean);
```

7. Input/Output Routines

The IRIS Graphics Library supports three classes of input devices:

- *valuators*, which return an integer value. For example, a dial and button box is a valuator. A mouse is a pair of valuators: one reports horizontal position and the other reports vertical position.
- *buttons*, which return a Boolean value: FALSE(0) when they are not pressed and TRUE(1) when they are pressed. Keys on an unencoded keyboard, buttons on a mouse, and switches on a dial and button box are all buttons.
- *other devices*, which return information about other system events. For example, the keyboard returns ASCII characters. See the *IRIS Programming Tutorial*, Chapter 5, for more information.

Within each class, individual devices have unique device numbers. Appendix A contains a header file, *device.h*, which defines symbolic names for all of the device numbers.

7.1 Polling and Queueing

Each input device has an associated value. If the input device is a button, the value is either 1 or 0. If the device is a valuator, such as a dial or the *x* position of the mouse, its value is an integer that indicates the position of the device.

A program can get the value from input devices in two ways:

- Polling immediately returns the value of a device. For example, `getbutton(LEFTMOUSE)` returns 1 if the left button of the mouse is down and returns 0 if it is up.

- Queueing uses an event queue to save changes in device values so the program can read them later.

In certain cases, using the input queue is better than polling. For example, `getbutton` returns only the current state of the button. Two successive calls to `getbutton` could both indicate that a button is up, although the user could have pressed and released the button between the two calls. If that happens, the button press/release event is lost. When a button is queued, the events are saved in the event queue and are available whenever the program queries it. In a drawing program where you may want to indicate a series of vertices and the system's calculations cannot keep up—queueing saves all the state changes so nothing is missed, even if the program is doing something else when the event happens.

Devices that are queued act as asynchronous devices, independent of the user process. Whenever a device that is queued changes state, an entry is made in the event queue. If a program reads the queue in a timely fashion, no events will be lost.

You can decide which devices, if any, to queue, and establish some rules about what constitutes a state change, or *event*, for those devices. By default, no devices are queued.

In addition to input routines, there are routines that control the characteristics of the peripheral input/output devices of the IRIS. These routines turn the keyboard click and the keyboard lights on and off; ring the keyboard bell; and control the lights and text on the dial and button box. See Tables 7-1, 7-2, and 7-3 for a listing of IRIS devices and their descriptions.

Devices	Description
MOUSE1	right mouse button
MOUSE2	middle mouse button
MOUSE3	left mouse button
MOUSEMIDDLE	middle mouse button
MOUSELEFT	left mouse button
SW0..SW31	32 buttons on dial and button box
AKEY..PADENTER	all the keys on the keyboard
LPENBUT	light pen button
BPAD0	pen stylus or button for digitizer tablet
BPAD1	button for digitizer tablet
BPAD2	button for digitizer tablet
BPAD3	button for digitizer tablet
MENUBUTTON	menu button

Table 7-1. Input Buttons

Devices	Description
MOUSEX	x valuator on mouse
MOUSEY	y valuator on mouse
DIAL0..DIAL8	dials on dial and button box
LPENX	x valuator on light pen
LPENY	y valuator on light pen
BPADX	x valuator on digitizer tablet
BPADY	y valuator on digitizer tablet
CURSORX	x valuator attached to cursor (usually MOUSEX)
CURSORY	y valuator attached to cursor (usually MOUSEY)
GHOSTX	x ghost valuator
GHOSTY	y ghost valuator
TIMER0..TIMER3	timer devices

Table 7-2. Input Valuators

Devices	Description
KEYBD	keyboard inputs ASCII characters
REDRAW	notifies process to redraw
MODECHANGE	indicates system change between single and double buffer mode
INPUTCHANGE	indicates change in input focus
PIECECHANGE	indicates change in exposed area of a window

Table 7-3. Other Devices

7.2 Initializing a Device

Several input devices, such as the dial and button box and the digitizing tablet, are attached to the IRIS via a serial port. Normally, if only one such input device is used, it is plugged into the bottom serial port on the back of the IRIS.

devport

Use `devport` if more than one input device is attached. *dev* indicates the device number, such as DIAL0 or BPADX. (See Table 7-2 for a list of input device numbers.) *portno* specifies the number of the serial port to which the device is connected.

Use `devport` on system startup before using any other peripheral devices.

```
devport(dev, portno)
Device dev;
long portno;

subroutine devpor(dev, portno)
integer*4 dev, portno

procedure devport(dev, portno: longint);
```

setvaluator

Valuators are single-value input devices. The value is a 16-bit integer. The horizontal and vertical motion of a mouse, or the turning of a dial, are valuators. `setvaluator` assigns an initial value *init* to a valuator. *min* and *max* are the minimum and maximum values the device can assume.

Note: Use of `setvaluator` under the window manager can restrict cursor movement.

```
setvaluator(val, init, min, max)
Device val;
short init, min, max;

subroutine setval(val, init, min, max)
integer*4 val, init, min, max

procedure setvaluator(val: Device; init, min, max: longint);
```

attachcursor

The cursor reflects the position of an input device that you (the user) control (see Chapter 6, Display and Color Modes). Since the cursor moves in two dimensions, any two valuators can determine its position. While it is customary to use the horizontal and vertical motion of the mouse to control the cursor, there is no restriction on which valuators you use. If desired, you can control the cursor using two dials on the optional dial and button box.

`attachcursor` attaches the cursor to the movement of two valuators. Its arguments are two valuator device numbers *(vx,vy)*. The first valuator determines the horizontal motion of the cursor; the second valuator determines its vertical motion.

```
attachcursor(vx, vy)
Device vx, vy;

subroutine attach(vx, vy)
integer*4 vx, vy

procedure attachcursor(vx, vy: Device);
```

7.3 Polling a Device

You can poll a device to determine its current state.

getvaluator

`getvaluator` determines the values of the valuators. You can poll any valuator, regardless of whether or not it is queued. The argument to `getvaluator` is a valuator device number *(val)*. Its value reflects the current state of the device.

```
long getvaluator(val)
Device val;

integer*4 function getval(val)
integer*4 val

function getvaluator(val: Device): longint;
```

getbutton

`getbutton` polls a button and returns its current state. Its argument is the number of the device you want to poll. `getbutton` returns either TRUE(1) or FALSE(0). TRUE(1) indicates the button is pressed.

```
Boolean getbutton(num)
Device num;

logical function getbut(num)
integer*4 num

function getbutton(num: Device): longint;
```

getdev

`getdev` polls up to 128 valuators and buttons at one time. You specify the number of devices you want to poll *(n)* and an array of device numbers *(devs)*. (See Tables 7-1, 7-2, and 7-3 for listings of device numbers.) *vals* returns the state of each specified device.

```
getdev(n, devs, vals)
long n;
Device *devs;
short *vals;

subroutine getdev(n, devs, vals)
integer*4 n
integer*2 devs(n), vals(n)

procedure getdev(n: longint; var devs: Short; var vals:
     Short);
```

7.4 The Event Queue

Input devices can make entries in the event queue. Each entry includes the device number and a device value. `qdevice` enables queueing of events from an input device. `unqdevice` indicates that a device is no longer queued. `isqueued` tells you if a specific device is queued.

qdevice

`qdevice` queues the specified device *(dev)*; for example, a keyboard, button, or valuator. The argument of `qdevice` is a device number. Each time the device changes state, an entry is made in the event queue. The maximum number of queue entries is 50.

```
qdevice(dev)
Device dev;

subroutine qdevic(dev)
integer*4 dev

procedure qdevice(dev: Device);
```

unqdevice

Use `unqdevice` to disable the queueing of events from a device. If the device has recorded events in the queue that have not been read, those events remain in the queue. (Use `qreset` to flush the event queue.)

```
unqdevice(dev)
Device dev;

subroutine unqdev(dev)
integer*4 dev

procedure unqdevice(dev: Device);
```

isqueued

`isqueued` indicates whether a specific device is enabled for queueing. It returns a Boolean value. TRUE(1) indicates that the device is enabled for queueing; FALSE(0) indicates that the device is not.

```
boolean isqueued(dev)
device dev;

logical function isqueue(dev)
integer (dev)

function isqueued(dev: Device): Boolean;
```

noise

Some valuators are noisy; that is, they report small fluctuations in value, indicating movement when no event has occurred. `noise` allows you to set a lower limit on what constitutes a move. The value of a noisy valuator *v* must change by at least *delta* before the motion is significant. `noise` determines how often queued valuators make entries in the event queue. For example, `noise(v,5)` means that valuator *v* must move at least five units before a new queue entry is made.

```
noise(v, delta)
Device v;
short delta;

subroutine noise(v, delta)
integer*4 v, delta

procedure noise(v: Device; delta: longint);
```

tie

You can tie a queued button to one or two valuators so that whenever the button changes state, the system records the button change and the current valuator position in the event queue. `tie` takes three arguments: a button *b* and two valuators *v1* and *v2*. Whenever the button changes state, three entries are made in the queue that record the current state of the button and the current position of each valuator. You can tie one valuator to a button by making *v2* equal to zero. You can untie a button from valuators by making both *v1* and *v2* equal to zero.

```
tie(b, v1, v2)
Device b, v1, v2;

subroutine tie(b, v1, v2)
integer*4 b, v1, v2

procedure tie(b, v1, v2: Device);
```

qenter

`qenter` creates event queue entries. It places entries directly into the event queue. `qenter` takes two 16-bit integers, *qtype* and *val*, and enters them into the event queue.

```
qenter(qtype, val)
short qtype, val;

subroutine qenter(qtype, val)
integer*4 qtype, val

procedure qenter(qtype, val: longint);
```

Three routines read the event queue: `qtest`, `qread`, and `blkqread`.

qtest

`qtest` returns the device number of the first entry in the event queue; if the queue is empty, it returns zero. `qtest` always returns immediately to the caller, and makes no changes to the queue.

```
long qtest()

integer*4 function qtest()

function qtest: longint;
```

qread

`qread`, as `qtest`, returns the device number of the first entry in the event queue. However, if the queue is empty, it waits until an event is added to the queue. `qread` returns the device number, writes the data part of the entry into *data*, and removes the entry from the queue.

```
long qread(data)
short *data;

integer*4 function qread(data)
integer*2 data

function qread(var data: Short): longint;
```

blkqread

`blkqread` returns multiple queue entries. Its first argument, *data*, is an array of short integers, and its second argument, *n*, is the size of the array data. `blkqread` returns the number of queue entries read and *data* is filled alternately with device numbers and device values. Note that the number of entries read is at most *n/2*.

`blkqread` can provide more efficient network communication between an IRIS terminal and a host computer. You can also use it when only the last entry in the event queue is of interest (e.g., when a user-defined cursor is being dragged across the screen and only its final position is of interest).

```
long blkqread(data, n)
short *data
short n;

integer*4 function blkqre(data, n)
integer*2 data(*)
integer*4 n

function blkqread(var data: Short; n:
    longint): longint;
```

qreset

`qreset` removes all entries from the queue and discards them.

```
qreset()

subroutine qreset

procedure qreset;
```

7.5 Controlling Peripheral Input/Output Devices

In addition to routines that poll and queue input devices, there are routines that control the characteristics and behavior of the IRIS's peripheral input/output devices. For example, some of these routines turn the keyboard click on and off (`clkon` and `clkoff`), or set the keyboard bell. You set these controls to your preference or needs.

clkon

`clkon` turns on the keyboard click.

```
clkon()

subroutine clkon

procedure clkon;
```

clkoff

`clkoff` turns off the keyboard click.

```
clkoff()

subroutine clkoff

procedure clkoff;
```

lampon

`lampon` and `lampoff` control the four lamps on the keyboard. Each 1 in the four lower-order bits of the *lamps* argument to `lampon` turns on the corresponding keyboard lamp.

```
lampon(lamps)
char lamps;

subroutine lampon(lamps)
integer*4 lamps

procedure lampon(lamps: longint);
```

lampoff

`lampoff` turns off any or all of the keyboard lamps. Each 1 in the four lower-order bits of the *lamps* argument to `lampoff` turns off the corresponding keyboard lamp.

```
lampoff(lamps)
char lamps;

subroutine lampof(lamps)
integer*4 lamps

procedure lampoff(lamps: longint);
```

ringbell

`ringbell` rings the keyboard bell.

```
ringbell()

subroutine ringbe

procedure ringbell;
```

setbell

`setbell` sets the duration of the keyboard bell: 0 is off; 1 is a short beep; and 2 is a long beep.

```
setbell(mode)
char mode;

subroutine setbel(mode)
integer*4 mode

procedure setbell(mode: longint);
```

dbtext

`dbtext` writes text to the LED display in a dial and button box. The string *str* must be eight or fewer uppercase characters.

```
dbtext(str)
char *str;

subroutine dbtext(str)
character*(8) str

procedure dbtext(varstr: Byte);
```

setdblight

`setdblights` controls the 32 lighted switches on a dial and switch box. For example, to turn on switches 3 and 7, the third and seventh bits to the right of *mask* must be *(1<<3)|(1<<7)*.

```
setdblights(mask)
long mask;

subroutine setdbl(mask)
integer*4 mask

procedure setdblights(mask: longint);
```

7.6 Special Devices

There are four types of special devices: keyboard, timer, cursor, and ghost devices.

7.6.1 Keyboard Devices

The keyboard device returns ASCII values that correspond to the keys typed on the keyboard. The device interprets keyboard movements in the standard manner, e.g., reports an event only on a downstroke, taking into account the 'crtl' and 'shift' keys.

7.6.2 Timer Devices

The timer devices record an event every 60th of a second. You can use a timer device to synchronize a graphics program with a real clock. To record events less frequently, use `noise`. For example, if you call

```
noise (TIMER0, 30)
```

only every 30th event is recorded, so an event queue entry is made each half second.

7.6.3 Cursor Devices

The cursor devices are pseudo devices that are equivalent to the valuators currently attached to the cursor. (See `attachcursor` for more information.)

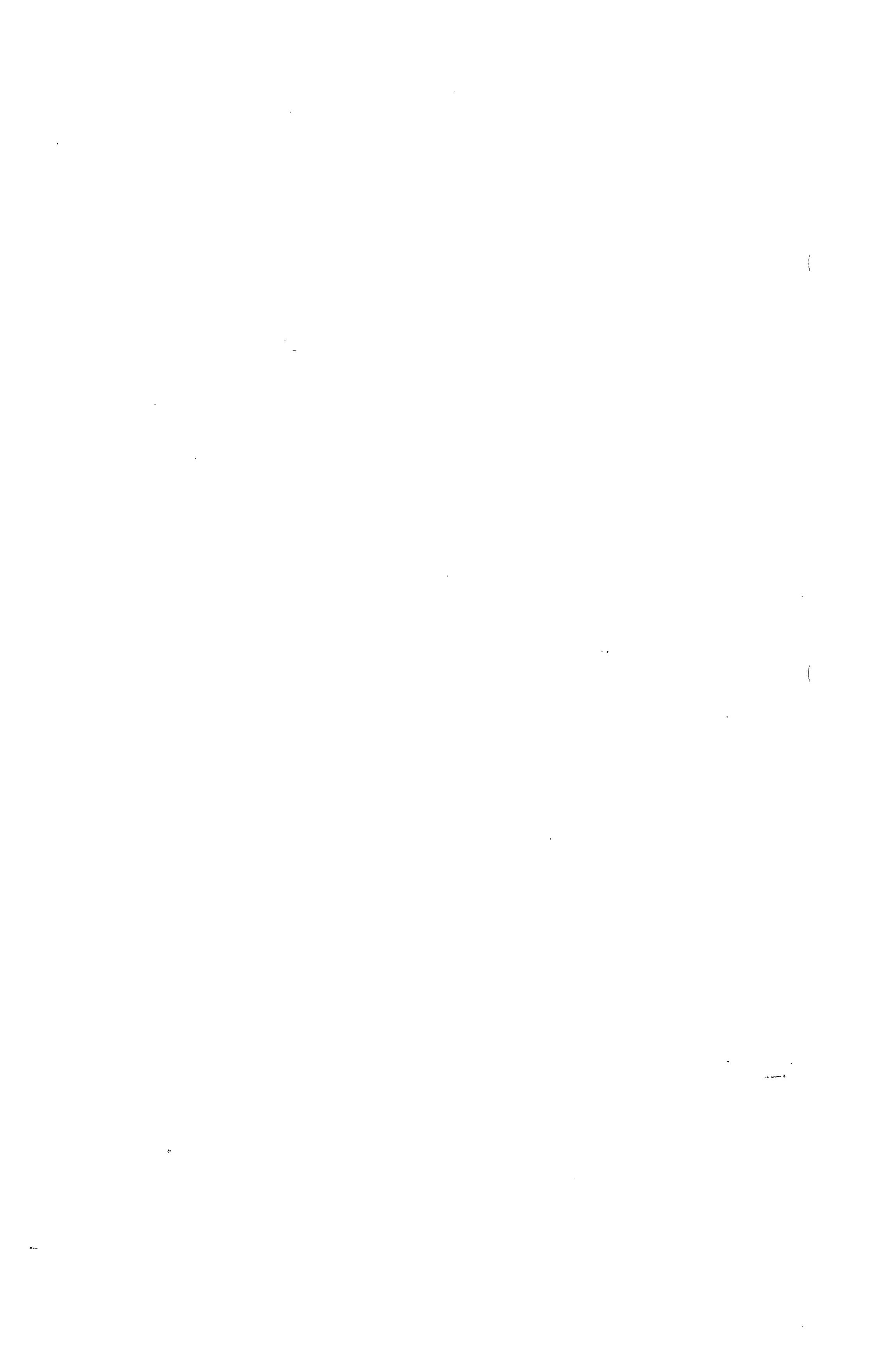
7.6.4 Ghost Devices

Ghost devices, GHOSTX and GHOSTY, don't correspond to any physical devices, although they can be used to change a device under program control. For example, to drive the cursor from software, use `attachcursor(GHOSTX,GHOSTY)` to make the cursor position depend on the ghost devices. Then use `setvaluator` on GHOSTX and GHOSTY to move the cursor.

7.6.5 Window Manager Devices

The following devices can only be used within the context of the window manager (see *Using mex, the IRIS Window Manager*).

REDRAW	the window manager inserts a redraw token each time the window needs to be redrawn; e.g., the user can modify the size or shape of the window.
MODECHANGE	when the process changes from single buffer mode to double buffer mode, or vice versa.
INPUTCHANGE	indicates a change in the input focus.
PIECECHANGE	indicates that the window is the same size and in the same position, but that a portion has been exposed; e.g., when a window that obscures the current window moves, exposing more of the current window.



8. Graphical Objects

It is sometimes convenient to group together a sequence of drawing routines and give it an identifier. The entire sequence can then be repeated with a single reference to the identifier rather than by repeating all the drawing routines. On the IRIS such sequences are called *graphical objects*; on other systems they are sometimes known as display lists. A graphical object is a list of graphics primitives (drawing routines) to display. For example, a drawing of an automobile can be viewed as a compilation of smaller drawings of each of its parts: windows, doors, wheels, etc. Each part (e.g., a wheel) might be a graphical object—a series of `move`, `draw`, and `poly` routines.

To make the automobile a graphical object, you would first define objects that draw its parts—a wheel object, a door object, a body object, and so on. The automobile object would be a series of calls to the part objects, which together with appropriate rotation, translation, and scale routines, would put all the parts in their correct places.

8.1 Defining An Object

You define and name objects with `makeobj`. When you call `makeobj`, the IRIS defines an object. Its argument *(obj)* is a 31-bit integer, which is the object's numeric identifier. When `makeobj` executes, the IRIS enters the object's numeric identifier into a symbol table and allocates memory for its list of drawing routines. This opens a new, empty object to which you can add drawing routines.

When you open an object for editing, drawing routines are not executed and drawn on the screen, but are added to the list until `closeobj` is called.

Thus, a graphical object is a list of primitive drawing routines to be executed. Drawing the display list consists of executing each of the routines in it from beginning to end. There is no flow control, such as looping, iteration, or condition texts (except for `bbox2`, see below).

Note: Not all Graphics Library routines can be put in a display list. A general rule is that drawing routines go into display lists, and Graphics Library routines that return values do not. If you have a question about a particular routine, check the manual page in the *Reference Guide.*

makeobj

`makeobj` defines a graphical object. It takes one argument, a 31-bit integer that is associated with the object. If *obj* is the number of an existing object, the contents of that object are deleted.

When `makeobj` executes, the object number is entered into a symbol table and memory is allocated for a display list. Subsequent graphics routines are compiled into the display list instead of executing.

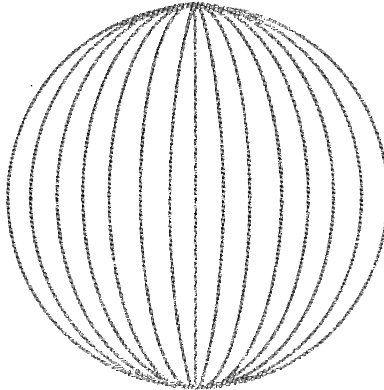
```
makeobj (obj)
Object obj;

subroutine makeob(obj)
integer*4 obj

procedure makeobj (obj: Object);
```

closeobj

`closeobj` terminates the object definition and closes the open object. All the routines in the graphical object between `makeobj` and `closeobj` are part of the object definition.

```
closeobj()

subroutine closeo

procedure closeobj;
```

Figure 8-1 shows an object definition of a simple shape named sphere, and the figure it draws when called.



```
makeobj(sphere=genobj());
for (phi=0; phi≤PI; phi+=PI/9){
   for (theta=0; theta≤2*PI; theta+=PI/18){
      x=sin(theta) * cos(phi);
      y=sin(theta) * sin(phi);
      z=cos(theta);
      if (theta==0)move(x,y,z);
      else draw (x,y,z);
   }
}
closeobj ( );
```

The sphere above is defined as a *graphical object. makeobj* creates a new object containing Graphics Library routines between *makeobj* and *closeobj.*

Figure 8-1. Object Definition

If you specify a numeric identifier that is already in use, the IRIS replaces the existing object definition with the new one. To ensure your object's numeric identifier is unique, use `isobj` and `genobj`.

isobj

`isobj` tests whether there is an existing object with a given numeric identifier. Its argument, *obj*, specifies the desired numeric identifier. `isobj` returns TRUE(1) if an object exists with the specified numeric identifier and FALSE(0) if none exists.

```
long isobj(obj)
Object obj;

logical function isobj(obj)
integer*4 obj

function isobj(obj: Object): longint;
```

genobj

`genobj` generates a unique numeric identifier. It does not generate any defined numeric identifiers. `genobj` is useful in naming objects when it is impossible to determine an undefined numeric identifier.

```
Object genobj()

integer*4 function genobj()

function genobj: Object;
```

delobj

`delobj` deletes an object. It frees all memory storage associated with the object. The numeric identifier is undefined until it is reused to define a new object. The system ignores calls to deleted or undefined objects.

```
delobj(obj)
Object obj;

subroutine delobj(obj)
integer*4 obj

procedure delobj(obj: Object);
```

8.2 Using Objects

callobj

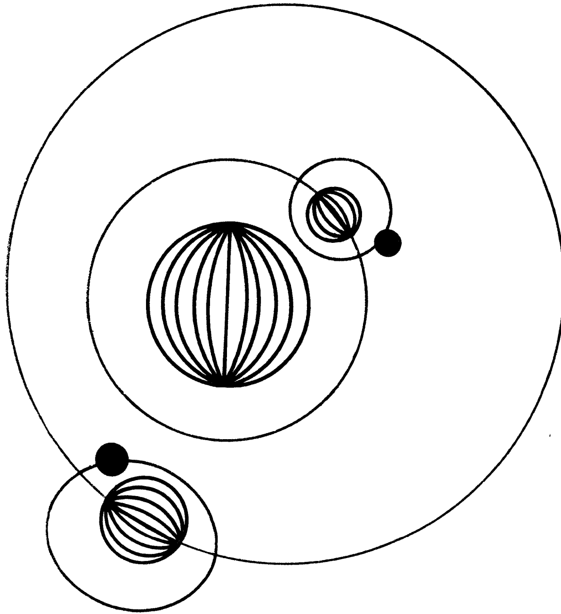
Once you define an object, use `callobj` to draw it on the screen. Its argument, *obj*, takes the numeric identifier of the object you want to draw.

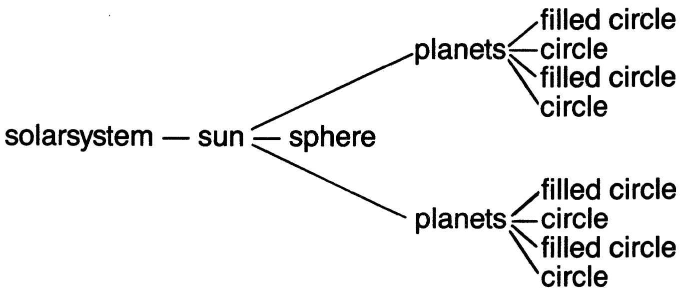
```
callobj(obj)
Object obj;

subroutine callob(obj)
integer*4 obj

procedure callobj(obj: Object);
```

Use `callobj` to call one object from inside another object. You can draw more complex pictures when you use a hierarchy of simple objects. For example, the program below uses a single `callobj(pearls)` to draw the object, a string of pearls, by calling the previously defined object `pearl` seven times.

```
Object pearl = 1, pearls = 2

makeobj(pearl);
    color(BLUE);
    for(angle=0; angle<3600; angle=angle+300) {
        rotate(300, 'y');
        circ(0.0, 0.0, 1.0);
    }
closeobj();


makeobj(pearls);
    for(i=0; i<7; i=i+1) {
        translate(2.0, 0.0, 0.0);
        color(i);
        callobj(pearl);
    }
closeobj();
```

Figure 8-2 shows another example using simple objects to build more complex ones. It defines a solar system as a hierarchical object. Calling one object, e.g., `solarsystem`, draws all the other objects named in its definition (the sun, the planets, and their orbits).





Solarsystem, a complex object, is defined hierarchically, as shown in the tree diagram. Branches in the tree represent *callobj* routines.

Figure 8-2. Hierarchical Objects

The IRIS does not save global attributes before `callobj` takes effect. Thus, if an attribute, such as color, changes within an object, the change can affect the caller as well. When needed, use `pushattributes` and `popattributes` to preserve global attributes across `callobj`.

When a complex object is called, the system draws the whole hierarchy of objects in its definition. For example, in Figure 8-2, because the system draws the whole object `solarsystem`, it can draw objects that are not visible in the viewport. `bbox2` determines whether or not an object is within the viewport, and whether it is large enough to be seen.

bbox2

`bbox2` performs the graphical functions known as *pruning* and *culling*. Culling determines which parts of the picture are less than the minimum feature size, and thus too small to draw on the screen. Pruning calculates whether an object is completely outside the viewport.

`bbox2` is a conditional test that you can position in a graphical object. If the conditions indicate that the same part of the bounding box would appear on the screen (not pruned) and was large enough (not culled), then the execution of a graphical object would proceed normally. During execution, if one of these two tests fails, the remaining routines in the object are ignored. `bbox2` optimizes your application by avoiding the execution of graphics that may be clipped or reduced to insignificant size.

`bbox2` takes as its arguments an object space bounding box *(x1, y1, x2, y2)* in coordinates and minimum horizontal and vertical feature sizes *(xmin, ymin)* in pixels. The IRIS calculates the bounding box, transforms it to screen coordinates, and compares it with the viewport. If the bounding box is completely outside the viewport, the routines between `bbox2` and the end of the object are ignored. If the bounding box is within the viewport, the system compares it with the minimum feature size. If it is too small in both the *x* and *y* dimensions, the rest of the routines in the object are ignored. Otherwise, the system continues to execute the object.

```
bbox2(xmin, ymin, x1, y1, x2, y2)
Screencoord xmin, ymin;
Coord x1, y1, x2, y2;

subroutine bbox2(xmin, ymin, x1, y1, x2, y2)
integer*4 xmin, ymin
real x1, y1, x2, y2

procedure bbox2(xmin, ymin: longint; x1, y1, x2, y2: Coord)
```

Figure 8-3 shows some of the objects within `solarsystem` juxtaposed to specified bounding boxes. The bounding boxes can perform pruning to determine what objects will be partially in the viewport.

8.3 Object Editing

You can change an object by editing it. Editing requires you to identify and locate the drawing routines that you want to change. You use two types of routines when you edit an object:

- *editing routines*, which add, remove, or replace drawing routines
- *tag routines*, which identify locations of drawing routines within an object

In the IRIS 3000 series, it is more efficient to define your own graphical objects or to use fast immediate mode, than it is to constantly edit graphical objects. See Appendix G, Fast Immediate Mode, for detailed information.

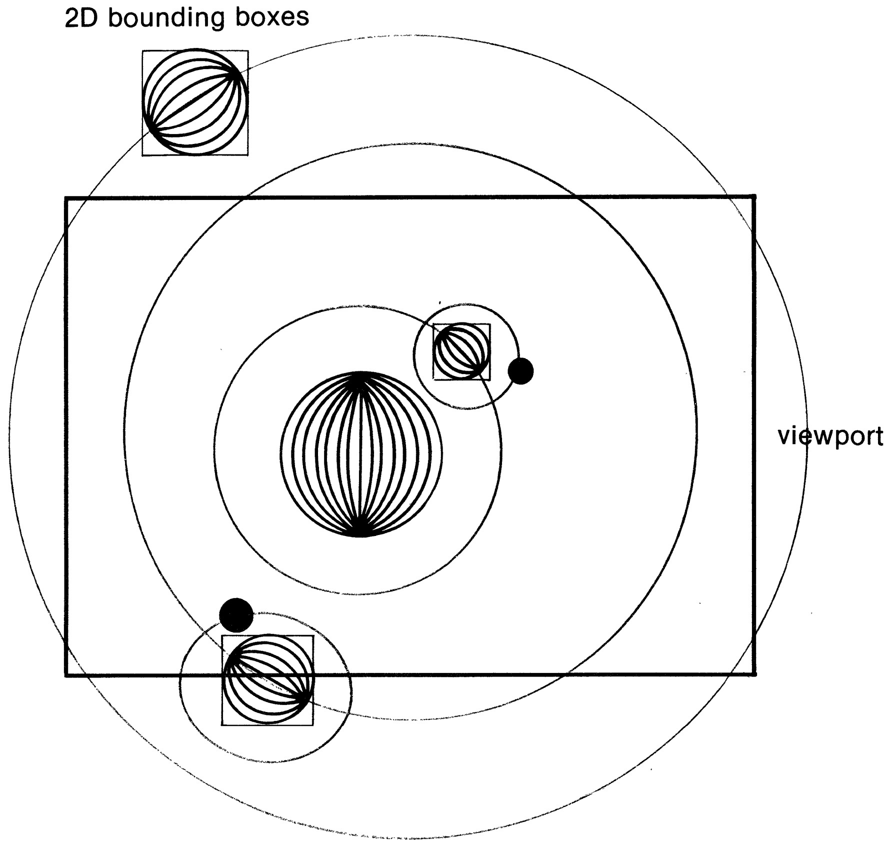
If you have to edit graphical objects, use your own display lists. The editing routines that follow are for compatibility with previous products.

editobj

`editobj` opens an object for editing. A pointer acts as a cursor that appends new routines. The pointer is initially set to the end of the object. The system appends graphics routines to the object until either a `closeobj` or a pointer positioning routine `(objdelete, objinsert,` or `objreplace`) executes.

The IRIS executes the editing routines following `editobj`. Use `closeobj` to terminate your editing session. If you specify an undefined object, an error message appears.

```
editobj(obj)
Object obj;

subroutine editob(obj)
integer*4 obj

procedure editobj(obj: Object);
```



Bounding boxes are computed to determine which objects are outside the screen viewport. If the bounding box is entirely outside the viewport, the rest of the object display list is not traversed. The sphere in the bounding box that lies partially within the viewport is drawn and clipped to the edge of the viewport.

Figure 8-3. Bounding Boxes

getopenobj

To determine if any objects are open for editing, use `getopenobj`. If an object is open, it returns the object's numeric identifier. It returns -1 if no objects are open. Figure 8-4 illustrates object editing.

```
Object getopenobj()

integer*4 function getope()

function getopenobj: Object;
```

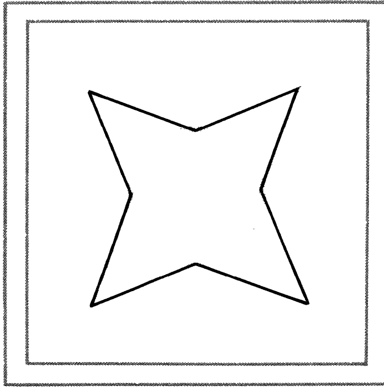
8.3.1 Identifying Display List Items with Tags

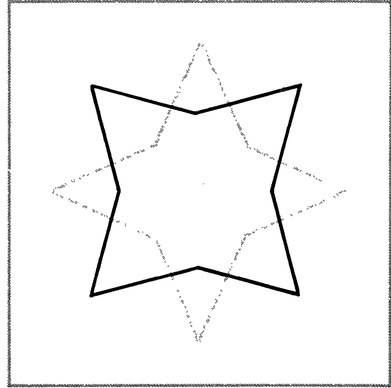
Once you define an object, you need a method of finding locations that may require changes. You can use tags in the same manner you would use a bookmark, i.e., to identify places you may need to locate. Tags locate display list items that you want to edit. Editing routines require tag names as arguments.

maketag

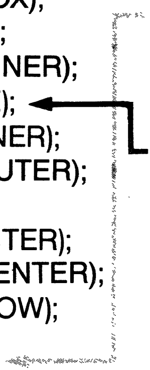
Use tags to mark display list items you may want to change. `maketag` explicitly tags routines. You specify a 31-bit numeric identifier and the system places a marker between two list items. You can use the same tag name in different objects.

```
maketag(t)
Tag t;

subroutine maketa(t)
integer*4 t

procedure maketag(t: Tag);
```





```
makeobj (star);
  color (GREEN);
  maketag (BOX);
  recti (1,1,9,9);
  maketag (INNER);
  color (BLUE);
  poly2i (8,INNER);
  maketag (OUTER);
  color (RED);
  poly2i (8,OUTER);
  maketag (CENTER);
  color (YELLOW);
  pnt2i (5,5);
closeobj ( );
```



```
editobj (star);
  circi (1,5,5);
  objinsert (BOX);
  recti (0,0,10,10);
  objreplace (INNER);
  color (BLACK);
  delete (OUTER,
          CENTER);
closeobj ( );
```

```
makeobj (star);
  color (GREEN);
  maketag (BOX);
  recti(0,0,10,10);
  recti (1,1,9,9)
  maketag (INNER);
  color (BLACK);
  poly2i (8,INNER);
  maketag (OUTER);
  tag (CENTER);
  color (YELLOW);
  pnt2i (5,5);
  circi (1,5,5);
closeobj ( );
```

The object definition on the left is being edited. Arrows indicate the movements of the editing cursor. The resulting object definition is on the right. The figures above show the object as it would be drawn before and after editing.

Figure 8-4. Editing an Object Definition

istag

`istag` tells whether a given tag is in use within the current open object. `istag` returns TRUE(1) if the tag is in use, and FALSE(0) if it is not. The result is undefined if there is no current open object.

```
Boolean istag(t)
Tag t;

logical function istag(t)
integer*4 t

function istag(t: Tag): longint;
```

gentag

`gentag` generates a unique integer to use as a tag within the current open object.

```
Tag gentag()

integer*4 function gentag()

function gentag: Tag;
```

deltag

`deltag` deletes tags from the object currently open for editing. Remember, you cannot delete the special tags STARTTAG and ENDTAG.

```
deltag(t)
Tag t;

subroutine deltag(t)
integer*4 t

procedure deltag(t: Tag);
```

newtag

`newtag` also adds tags to an object, but uses an existing tag to determine its relative position within the object. `newtag` defines a new tag that is offset beyond the other tag by the number of lines given in its argument *offset*.

```
newtag(new, old, offset)
Tag new, old;
long offset;

subroutine newtag(new, old, offset)
integer*4 new, old, offset

procedure newtag(new, old: Tag; offset: Offset);
```

`STARTTAG` is a predefined tag that goes before the very first item in the list; it marks the beginning of the list. `STARTTAG` does not have any effect on drawing or modifying the object. Use it only to return to the beginning of the list. `ENDTAG` is a predefined tag that is positioned after the last item on the list; it marks the end of the list. Like `STARTTAG`, `ENDTAG` does not have any effect on drawing or modifying the object. Use it to find the end of the graphical object. When you call `makeobj` to define a list, `STARTTAG` and `ENDTAG` automatically appear. You cannot delete these tags. When an object is opened for editing, there is a pointer at `ENDTAG`, just after the last routine in the object. To perform edits on other items, refer to them by their tags.

8.3.2 Inserting, Deleting, and Replacing within Objects

objinsert

objinsert `objinsert` adds routines to an object at the location specified in *t*. `objinsert` takes a tag as an argument, and positions an editing pointer on that tag. The system inserts graphics routines immediately after the tag. To terminate the insertion, use `closeobj` or another editing routine (`objdelete`, `objinsert`, or `objreplace`).

```
objinsert(t)
Tag t;

subroutine objins(t)
integer*4 t

procedure objinsert(t: Tag);
```

objdelete

`objdelete` removes routines from the current open object. It removes everything between *tag1* and *tag2*—it deletes routines and other tag names. For example, `objdelete(STARTTAG,ENDTAG)` would delete every routine, except the tags `STARTTAG` and `ENDTAG` themselves.

The IRIS ignores `objdelete` if no object is open for editing. This routine leaves the pointer at the end of the object after it executes.

```
objdelete(tag1, tag2)
Tag tag1, tag2;

subroutine objdel(tag1, tag2)
integer*4 tag1, tag2

procedure objdelete(tag1, tag2: Tag);
```

objreplace

`objreplace` combines the functions of `objdelete` and `objinsert`. It provides a quick way to replace one routine with another that occupies the same amount of display list space. Its argument is a single tag, *t*. Graphics routines that follow `objreplace` overwrite existing routines until a `closeobj` or editing routine `(objinsert, objreplace,` or `objdelete)` terminates the replacement.

Note: `objreplace` requires that the new routine be exactly the same length in characters as the previous one. Use `objdelete` and `objinsert` for more general replacement.

```
objreplace(t)
Tag t;

subroutine objrep(t)
integer*4 t

procedure objreplace(t: Tag);
```

8.3.3 Example

The following is an example of object editing. An object `star` is defined:

```
makeobj(star);
        color(GREEN);
        maketag(BOX);
        recti(1, 1, 9, 9);
        maketag(INNER);
        color(BLUE);
        poly2i(8, Inner);
        maketag(OUTER);
        color(RED);
        poly2i(8, Outer);
        maketag(CENTER);
        color(YELLOW);
        pnt2i(5, 5);
closeobj();
```

Then this object is edited with the following routine to give a modified object:

```
editobj(star);
        circi(1, 5, 5);
        objinsert(BOX);
        recti(0, 0, 10, 10);
        objreplace(INNER);
        color(GREEN);
closeobj();
```

The object resulting from the editing session is equivalent to an object defined by the following code.

```
makeobj(star);
        color(GREEN);
        maketag(BOX);
        recti(0, 0, 10, 10);
        recti(1, 1, 9, 9);
        maketag(INNER);
        color(GREEN);
        poly2i(8, Inner);
        maketag(OUTER);
        color(RED);
        poly2i(8, Outer);
        maketag(CENTER);
        color(YELLOW);
        pnt2i(5, 5);
        circi(1, 5, 5);
closeobj();
```

8.3.4 Object Memory Management

Editing can require large amounts of memory. `compactify`, `chunksize`, and `getmem` perform memory management tasks that you may need.

compactify

As memory is modified by the various editing routines, an open object can become fragmented and stored inefficiently. When the amount of wasted space becomes large, the IRIS automatically calls `compactify` during the `closeobj` operation. The routine allows you to perform the compaction explicitly. Unless you insert new routines in the middle of an object, compaction is not necessary.

Note: `compactify` uses a significant amount of computing time. Do not call it unless the amount of available storage space is critical.

```
compactify(obj)
Object obj;

subroutine compac(obj)
integer*4 obj

procedure compactify(obj: Object);
```

chunksize

If there is a memory shortage, you can use `chunksize` to allocate memory differently to an object. `chunksize` specifies the minimum amount of memory that the system allocates to an object. When you specify *chunk*, its size should vary according to the needs of the application. As the object grows, fixed-sized chunks of memory are added to it. There are cases where this can be inefficient. For example, if the objects are small and the chunks are large, most objects will consist of one large chunk, most of which is wasted space. On the other hand, if the objects are large and the chunks are small, there is an overhead associated in keeping track of the large number of chunks.

Only one chunk size is allowed per application and the default size is 1020 bytes. This can be changed once to a more suitable size after a call to `winopen`, `ginit`, `gbegin`, or `getport`, and before the first call to `makeobj`.

The chunk size must be large enough to hold the largest Graphics Library routine in the display list. If your chunk size is too small (perhaps you have polygons with a large number of sides) execution errors can occur.

Note: `chunksize` is a space optimization technique, and should be used only if memory is limited for the application. Determination of the best value for `chunksize` is a trial and error process.

```
chunksize(chunk)
long chunk;
```

```
subroutine chunks(chunk)
integer*4 chunk
```

```
procedure chunksize(chunk: longint);
```

getmem

To determine the amount of available memory, use `getmem`. On a terminal, it returns the amount of free physical memory. On a workstation with virtual memory up to 14 megabytes, it returns 14 megabytes less the amount in use.

```
long getmem()
```

```
integer*4 function getmem()
```

```
function getmem: longint;
```

callfunc

`callfunc` allows an arbitrary function call from within an object. When it executes in the object, the function *(fctn, nargs, arg1, arg2, ..., argn)* is called. `callfunc` is useful only for writing customized terminal programs. It cannot be called remotely, and should not be used on a workstation.

```
callfunc(fctn, nargs, arg1, arg2, ..., argn)
int (*fctn)()
long nargs, arg1, arg2, ..., argn;
```

Note: You can only use `callfunc` when programming in C.

9. Picking and Selecting

The previous chapters explain how to define objects in world coordinates so the system can draw them on the screen. This chapter discusses the reverse process: how to determine what routines define objects in a specified area of the screen. There are two ways to do this:

- `mapw` takes 2-D screen coordinates and identifies the corresponding 3-D world coordinates; or
- `pick` and `gselect`, which identify objects drawn in a specified 3-D area.

9.1 Mapping Screen Coordinates to World Coordinates

mapw

`mapw` takes a 2-D screen point and maps it onto a line in 3-D world space. Its argument *vobj* contains the viewing, projection, and viewport transformations that map the current displayed objects to the screen. `mapw` reverses these transformations and maps the screen coordinates back to world coordinates. It returns two points (*wx1*, *wy1*, *wz1*) and (*wx2*, *wy2*, *wz2*), which specify the endpoints of the line. *sx* and *sy* specify the screen point to be mapped.

```
mapw(vobj, sx, sy, wx1, wy1, wz1, wx2, wy2, wz2)
Object vobj;
Screencoord sx, sy;
Coord *wx1, *wy1, *wz1, *wx2, *wy2, *wz2;

subroutine mapw(vobj, sx, sy, wx1, wy1, wz1, wx2, wy2, wz2)
integer*4 vobj, sx, sy
real wx1, wy1, wz1, wx2, wy2, wz2

procedure mapw(vobj: Object; sx, sy: longinit; var wx1, wy1,
    wz1, wx2, wy2, wz2: Coord);
```

mapw2

`mapw2` is the 2-D version of `mapw`. In 2D, the IRIS maps screen coordinates to world coordinates rather than to a line. Again, *vobj* contains the projection and viewing transformations that map the displayed objects to world coordinates; *sx* and *sy* define screen coordinates. *wx* and *wy* return the corresponding world coordinates. If the transformations in *vobj* are not 2-D (i.e., not orthogonal projections), the result is undefined.

```
mapw2(vobj, sx, sy, wx, wy)
Object vobj;
Screencoord sx, sy;
Coord *wx, *wy;

subroutine mapw2(vobj, sx, sy, wx, wy)
integer*4 vobj, sx, sy
real wx, wy

procedure mapw2(vobj: Object; sx, sy: longinit; var wx, wy:
    Coord);
```

9.2 Picking

You use picking mode to identify objects on the screen that appear near the cursor. To use picking effectively, your software must be structured in such a way that you can regenerate the picture on the screen whenever picking is required. When it is, set the system into picking mode; using `pick`, redraw the image on the screen, and finally, call `endpick`. The results of the pick appear in the buffer specified by `pick` and `endpick`.

While it is in picking mode, the IRIS does not draw anything on the screen. Instead, drawing routines that would have been drawn near the cursor cause hits to be recorded in the picking buffer in a manner described below. With one exception, all the standard drawing routines cause hits, including clear, points, lines, polygons, arcs, circles, curves, and patches. Raster objects, such as character strings and pixels drawn with `charstr`, do not cause hits, although `cmov` does. Thus, to be picked, the cursor must be near the lower-left corner of the string. Note also that since `readpixels` and `readRGB` are often preceded by `cmov`, these routines can appear to cause hits. See Figure 9-1.

To identify the object(s) on the screen that caused hits, a name stack is supported. The name stack is a stack of 16-bit names whose contents are controlled by `loadname`, `pushname`, `popname`, and `initname`. In picking mode, whenever one of the routines that alters the name stack is issued or whenever picking mode is exited by `endpick` (or `endselect`, which is described in a later section), the contents of the name stack is recorded if a hit occurred since the name stack was last altered.

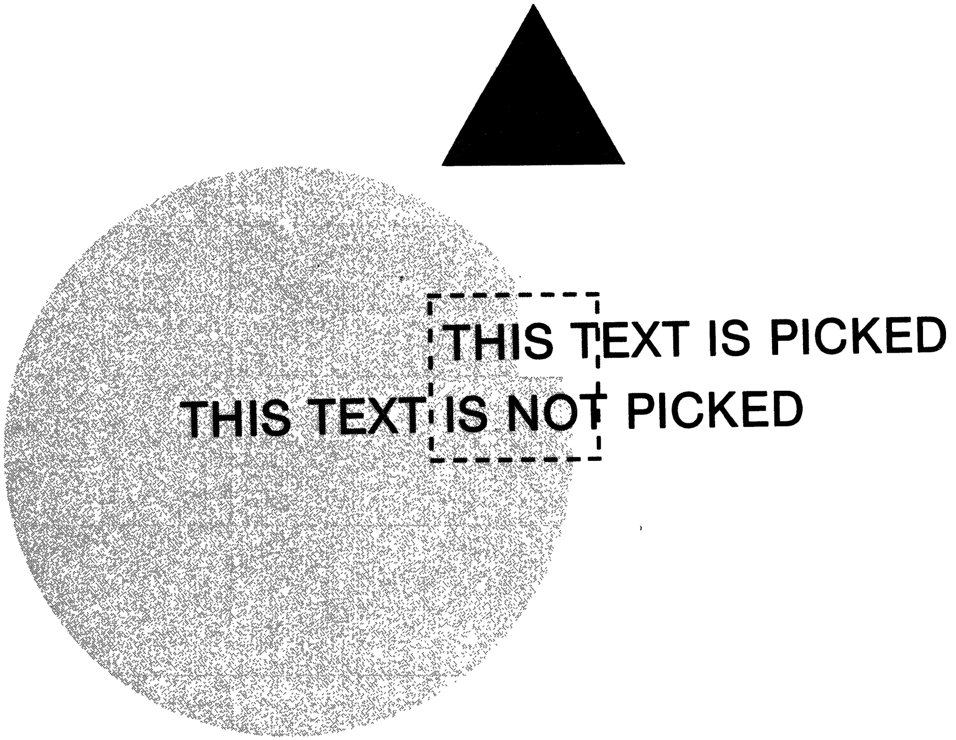
For example, suppose your application draws three widely spaced points on the screen, and you want to find out which one is closest to the cursor using picking mode. Your point-drawing code (that is executed both to draw points and to redraw them in a picking operation), might look something like this:

```
ortho(<ortho parameters>);[1]
loadname(0);
pnt(<point 0>);
loadname(1);
pnt(<point 1>);
loadname(2);
pnt(<point 2>);
```

When this code segment is executed in picking mode, if the cursor were near point 1, the buffer returned after `endpick` would contain the name 1; if it were near point 2, the buffer would contain the name 2; and if the cursor weren't near any of the points, an empty buffer would be returned.

---

1. Note that the complete specification for drawing the picture must be there, including any viewing and transformation routines.



In picking mode, you can identify the parts of an image that lie near the cursor. The cursor is shown as an arrow. The small box at the tip of the arrow is the *picking region*. The large shaded circle is picked. The text string whose origin is in the picking region is also picked. The shaded triangle and the other text string are not picked.

Figure 9-1. Picking

If hierarchical objects are drawn (say we draw a car with four instances of a wheel, and each wheel has five instances of a bolt, and we would like to pick an individual bolt from the picture), the name stack can be used effectively. We might have one piece of code to draw each wheel that contained the sequence:

```
pushname(0);
<draw bolt 0>
popname();
pushname(1);
<draw bolt 1>
popname();
        .
        .
        .
```

The car drawing code might look like this:

```
loadname(0);
<translate>
<draw wheel>
loadname(1);
<translate>
<draw wheel>
        .
        .
        .
```

Each hit on a bolt would occur with the name stack containing two names, the first of which is the wheel number and the second of which is the bolt number on that wheel. Deeper nesting of the hierarchy is obviously possible.

The names reported on hits are completely application dependent. Many drawing routines can occur between changes to the name stack, and if any of those routines cause a hit, the contents of the name stack is reported. Since the contents of the name stack is reported only when it changes, one hit will be reported no matter how many of the drawing routines actually drew something near the cusor. If more accuracy than this is required by the application, it must simply touch the name stack more often. In the code below, if all three points caused hits, three identical name stacks would be reported:

```
loadname(1);
pnt(-);
loadname(1);
pnt(-);
loadname(1);
pnt(-);
```

pick

`pick` puts the system in picking mode. The *numnames* argument to `pick` specifies the maximum number of values that the buffer can store. The graphical items that intersect the picking region are hits and store the contents of the name stack in *buffer*.

```
pick(buffer, numnames)
short buffer[];
long numnames;

subroutine pick(buffer, numnam)
integer*2 buffer(*)
integer*4 numnam

procedure pick(var buffer[0]: Short; numnames: longint);
```

9.2.1 Using the Name Stack

You maintain the name stack with `loadname`, `pushname`, `popname`, and `initnames`.

Each name in the name stack is 16 bits long. You can store up to 1000 names in a name stack. You can intersperse these routines with drawing routines, or you can insert them into object definitions. See Chapter 8, Graphical Objects, for a discussion of objects.

loadname

`loadname` puts *name* at the top of the name stack and erases what was there before.

```
loadname(name)
short name;

subroutine loadna(name)
integer*4 name

procedure loadname(name: longint);
```

pushname

`pushname` puts *name* at the top of the stack and pushes all the other names in the stack one level lower.

```
pushname(name)
short name;

subroutine pushna(name)
integer*4 name

procedure pushname(name: longint);
```

popname

`popname` discards the name at the top of the stack and moves all the other names up one level.

```
popname()

subroutine popnam

procedure popname;
```

initnames

`initnames` discards all the names in the stack and leaves the stack empty.

```
initnames()

subroutine initna

procedure initnames;
```

endpick

`endpick` takes the IRIS out of picking mode and returns the number of hits that occurred in the picking session. If it returns a positive number, the buffer stored all of the name lists. If it returns a negative number, the buffer was too small to store all the name lists; the magnitude of the returned number is the number of name lists that were stored.

buffer contains all of the name lists stored in picking mode, one list for each valid hit. The first value in each name list is the length of a name list. If a name stack is empty when a hit occurs, the first and only name in the list for that hit is '0'.

```
long endpick(buffer)
short buffer[];

integer*4 function endpic(buffer)
integer*2 buffer(*)

function endpick(var buffer: Short): longint;
```

gethitcode

`gethitcode` returns the global variable *hitcode*, which keeps a cumulative record of clipping plane hits. It does not change the hitcode value. The hitcode is a 6-bit number (see below), with 1 bit for each clipping plane.

5	4	3	2	1	0
far	near	top	bottom	right	left

```
long gethitcode()

integer*4 function gethit()

function gethitcode(): longint;
```

clearhitcode

`clearhitcode` clears the global variable *hitcode,* which records clipping plane hits in picking and selecting modes.

```
clearhitcode()

subroutine clearh

procedure clearhitcode;
```

9.2.2 Defining the Picking Region

Picking loads a projection matrix that makes the picking region fill the entire viewport. This picking matrix replaces the projection transformation matrix that is normally used when drawing routines are called. Therefore, you must restate the original projection transformation after `pick` to ensure the system maps the objects to be picked to the proper coordinates. If no projection transformation was originally issued, you must specify the default, `ortho2`. When the transformation routine is restated, the product of the transformation matrix and the picking matrix is placed at the top of the matrix stack. If you do not restate the projection transformation, picking does not work properly. Instead, the system typically picks every object, regardless of cursor position and picksize (including the viewport).

picksize

The default height and width of the picking region is 10 pixels centered at the cursor. You can change the picking region with `picksize`. *deltax* and *deltay* specify a rectangle centered at the current cursor position (the origin of the cursor glyph). (See Chapter 6, Display and Color Modes, Section 6.4 for a discussion of cursors.)

```
picksize(deltax, deltay)
short deltax, deltay;

subroutine picksi(deltax, deltay)
integer*4 deltax, deltay

procedure picksize(deltax, deltay: Short);
```

9.2.3 Example

The following program draws an object consisting of three shapes; then it loops, until the right mouse button is pressed. Each time the middle mouse button is pressed:

1. The system enters picking mode.
2. The system calls the object.
3. The system records hits for any routines that draw into the picking region.
4. The system prints out the contents of the picking buffer.

Note: When you call an object in picking mode, the screen does not change. Since the picking matrix is recalculated only when `pick` is called, the system exits and reenters picking mode to obtain new cursor positions.

■ C Program: PICKING

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

main()
{
    short namebuffer[50];
    long numpicked;
    short type, val, i, j, k;

    ginit();
    qdevice(MOUSE1);
    qdevice(MOUSE2);

    makeobj(1);
        color(RED);
        loadname(1);  /* load the name "1" on the name stack */
        rectfi(20,20,100,100);
        loadname(2);  /* load the name "2", replacing "1" */
        pushname(3);  /* push name "3", so the stack
                         has "3 2" */
        circi(50,500,50);
```

```
        loadname(4);  /* replace "3" with "4", so the stack
                         has "4 2" */
        circi(50,530,60);
    closeobj();

    color(BLACK);
    clear();
    callobj(1);  /* draw the object on the screen */

    while (1) {  /* loop until the right mouse button is pushed */
        type = qread(&val);
        if (val == 0)
        continue;
        switch (type) {
        case MOUSE1:  /* if the right mouse button is pushed,
                         the program exits */
            gexit();
            exit(0);
        case MOUSE2:  /* if the middle mouse button is pushed,
                         the IRIS enters pick mode */
            pushmatrix()
            pick(namebuffer, 50);
            /* restate the projection transformation for
            the object */
            ortho2(-0.5, XMAXSCREEN + 0.5, -0.5,
              YMAXSCREEN + 0.5);
            callobj(1);  /* call the object (no actual drawing
                   takes place) */
            numpicked = endpick(namebuffer);
                /* print out the number of hits and a name list
                for each hit */
                     popmatrix()
                        printf("hits: %d; ",numpicked);
            j = 0;
            for (i = 0; i < numpicked; i++) {
            printf(" ");
            k = namebuffer[j++];
            printf("%d ", k);
            for (;k; k--)
                printf("%d ", namebuffer[j++]);
            printf("|");
```

```
            }
            printf("\n");
        }
    }
}
```

■ FORTRAN Program: PICKING

```
#      include "fgl.h"
#      include "fdevice.h"
C
       INTEGER*2 NAMBUF(50)
       INTEGER*2 VAL, I, J, K, L
       INTEGER TYPE, NUMPIC
C
       CALL GINIT()
C
       CALL QDEVIC(MOUSE1)
       CALL QDEVIC(MOUSE2)
C
       CALL MAKEOB(1)
         CALL COLOR(RED)
C          load the name "1" on the name stack
         CALL LOADNA(1)
         CALL RECTFI(20,20,100,100)
C          load the name "2", replacing "1"
         CALL LOADNA(2)
C          push the name "3", so the stack has "3 2"
         CALL PUSHNA(3)
         CALL CIRCI(50,500,50)
C          replace "3" with "4", so the stack has "4 2"
         CALL LOADNA(4)
         CALL CIRCI(50,530,60)
       CALL CLOSEO()
C
       CALL COLOR(BLACK)
       CALL CLEAR()
C        draw the object on the screen
       CALL CALLOB(1)
C
C        loop until the left mouse button is pushed
```

```
100   CONTINUE
        TYPE = QREAD(VAL)
C         try again if the event was a button release
        IF (VAL .EQ. 0) GO TO 100
C         if the left mouse button is pushed, the program exits
        IF (TYPE .EQ. MOUSE1) THEN
          CALL GEXIT()
          GO TO 400
        END IF
C         if the middle mouse button is pushed, the IRIS
C         enters pick mode
    CALL PUSHMA()
        IF (TYPE .EQ. MOUSE2) THEN
          CALL PICK(NAMBUF,50)
C           restate the projection transformation for
C           the object
          CALL ORTHO2(-0.5, XMAXSC+0.5, -0.5, YMAXSC+0.5)
C           call the object (no actual drawing takes place)
          CALL CALLOB(1)
C           print out the number of hits and a name list for
C           each hit
          NUMPIC = ENDPIC(NAMBUF)
      CALL POPMAT()
          PRINT *, 'Hits: ', NUMPIC
          J = 1
          DO 300 I=1,NUMPIC
            K = NAMBUF(J)
            J = J + 1
            PRINT *,K
            DO 200 L=1,K
              PRINT *,'    ',NAMBUF(J)
              J = J + 1
200         CONTINUE
300       CONTINUE
        END IF
C       loop until the left mouse button is pushed
      GO TO 100
400   CONTINUE
      STOP
      END
```

When the program is run, there are five possible outcomes for each picking session (the circles can be picked together because they overlap):

- nothing is picked = "hits: 0;"
- the rectangle is picked = "hits: 1; 1 1 |"
- the first circle is picked = "hits: 1; 2 2 3 |"
- the second circle is picked = "hits: 1; 2 2 4 |"
- both the first and second circle are picked = "hits: 2; 2 2 3 | 2 2 4 |"

9.3 Selecting

Selecting is a more general mechanism than picking for identifying the routines that draw to a particular region. A selecting region is a 2-D or 3-D area of world space. When `gselect` turns on selecting mode, the region represented by the current viewing matrix becomes the selecting region. You can change the selecting region at any time by issuing a new viewing transformation routine. To use selecting mode:

1. Issue a viewing transformation routine that specifies the selecting region.
2. Call `gselect`.
3. Call the objects or routines of interest.
4. Exit selecting mode and look to see what was selected.

gselect

gselect turns on the selecting mode. gselect and pick are identical, except gselect allows you to create a viewing matrix in selecting mode.

numnames specifies the maximum number of values that the buffer can store. Names are 16-bit numbers, which you can store on the name stack. Each drawing routine that intersects the selecting region causes the contents of the name stack to be stored in *buffer*. The name stack and the hitcode routines are used in the same way as in picking.

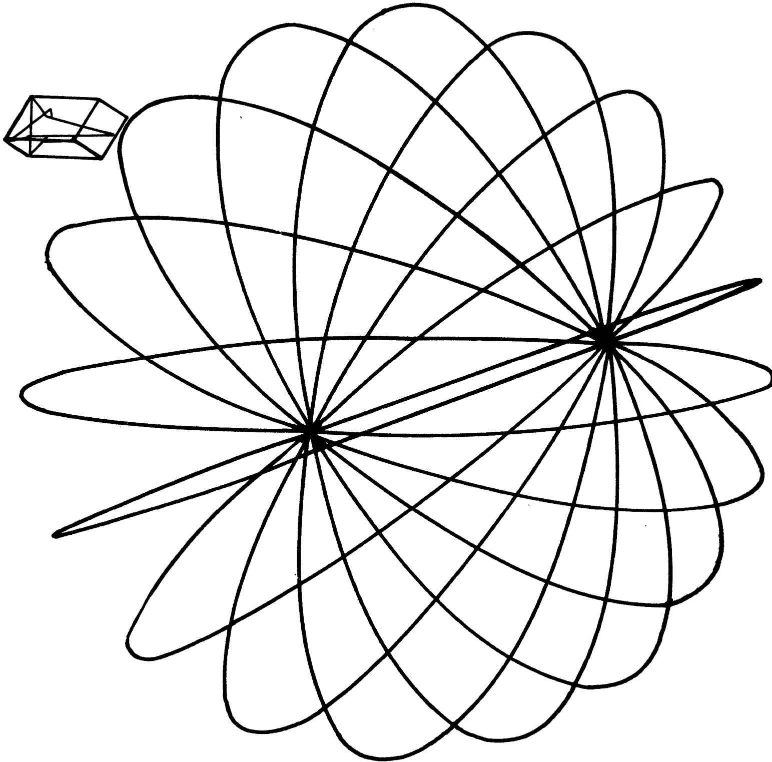
```
gselect(buffer, numnames)
short buffer[];
long numnames;

subroutine gselec(buffer, numnam)
integer*2 buffer(1)
integer*4 numnam

procedure gselect(var buffer[0]: Short; numnames: longint);
```

endselect

endselect turns off selecting mode. *buffer* stores any hits the drawing routines generate between gselect and endselect. Each name list represents the contents of the name stack when a routine was called that drew into the selecting region. endselect returns the number of name lists in *buffer*. If the number is negative, more routines drew into the selecting region than were specified by *numnames*.

```
long endselect(buffer)
short buffer[];

integer*4 function endsel(buffer)
integer*2 buffer(*)

function endselect(var buffer[MAXARRAY]:
    Short; size: longint);
```

Figure 9-2 shows a sample application of gselect using a rocket ship and a planet. The following program uses selecting to determine if the ship is colliding with the planet. The program calls a simplified version of the planet and draws a box representing the ship each time the left mouse button is pressed. The program prints 'CRASH' and exits when the ship collides with the planet.



In selecting mode, you can identify all routines that draw things in an arbitrary region of world space. Here, you can detect that the rocketship will crash into the planet by defining the selecting region as a box around the ship.

Figure 9-2. Sample Application of select

■ C Program: CRASH

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

#define PLANET 1

main()
{
    short type, val;
    register short buffer[50], cnt, i;
    float shipx, shipy, shipz;

    /* initialize the buffer to zeros */
    for (i = 0; i < 50; i++)
    buffer[i] = 0;

    ginit();

    qdevice(MOUSE3);
    color(BLACK);
    clear();
    color(RED);
    /* create the planet object */
    createplanet(PLANET);
    /* draw the planet on the screen */
    callobj(PLANET);

    while (1) {     /* loop until the left mouse button
            is pressed */
    type = qread(&val);
    if (val==0)
        continue;
    switch (type){
        case MOUSE3:
            /* set ship location to cursor location */
            shipz=0;
            shipx=getvaluator(MOUSEX);
            shipy=getvaluator(MOUSEY);
            /* draw the ship */
            color(BLUE);
            rect(shipx, shipy, shipx+20, shipy+10);
```

```
        /* specify the selecting region to be a box
        surrounding the rocket ship */

        pushmatrix();  /* save the current transformation
                   matrix */
        ortho(shipx,shipx+.05,shipy,shipy+.05,shipz-0.5,
             shipz+.05);

        /* clear the name stack */
        initnames();

        gselect(buffer, 50);  /* enter selecting mode */

        /* put "1" on the name stack to be saved if PLANET
        draws into the selecting region */
        loadname(1);
        pushname(2);

        /* call the planet object (no actual drawing
        takes place) */
        callobj(PLANET);

        /* exit selecting mode */
        cnt = endselect(buffer);
        popmatrix(); /* restore the original transformation
                 matrix */

        /* check to see if PLANET was selected */
        if (buffer[1]==1) {
            printf("CRASH\n");
            curson();
            gexit();
            }
    }
    }
    gexit();
}

createplanet(x)
{
    makeobj(x);
```

```
    circfi(200, 200, 20);
    closeobj();
}
```

■ FORTRAN Program: CRASH

```
      #include "fgl.h"
      #include "fdevice.h"

      integer*4 type,cnt
      integer*2 val
      integer*2 buffer(50)
      real shipx,shipy,shipz
      data iplant/1/

      do 10 i=1,50
     buffer(i) = 0
 10   continue

      call ginit
      call qdevic(MOUSE3)
      call qdevic(MOUSE1)
      call color(BLACK)
      call clear
      call color(RED)

      call create(iplant)
      call callob(iplant)

 15   continue
      type = qread(val)
c*
```

```
c*    loop until the left mouse button is pressed
c*
      if (val .eq. 0) go to 15
      if (type .eq. MOUSE3) then
     shipz=0
     shipx=getval(MOUSEX)
     shipy=getval(MOUSEY)

     call color(BLUE)
         call rect(shipx,shipy,shipx+20,shipy+10)

     call pushma
     call ortho(shipx,shipx+.05,shipy,shipy+.05,shipz-.05,
           shipz+.05)
     call initna

     call gselect(buffer,50)

     call loadna(1)
     call pushna(2)
     call callob(iplant)

     cnt=endsel(buffer)
     call popmat

     if (buffer(2) .eq. 1) then
        print *,'CRASH'
        call color(BLACK)
        call clear
        call curson()
        call gexit
         else
        go to 15
         end if
      else if (type .eq. RIGHTM)then
           GO TO 999
          else
        go to 15
      end if
```

```
999   continue
      call gexit
      stop
      end
      subroutine create(x)
      integer x
      call makeob(x)
      call circfi(200,200,20)
      call closeo
      return
      end
```

10. Geometry Pipeline Feedback

The IRIS feedback facility provides access to the Geometry Pipeline for matrix multiplication and geometric computing. In feedback mode, the system stores the data generated by the Geometry Pipeline in a buffer, rather than send it to the raster display subsystem. You can examine the buffer to see how the data was transformed. Two conditions exist in feedback mode:

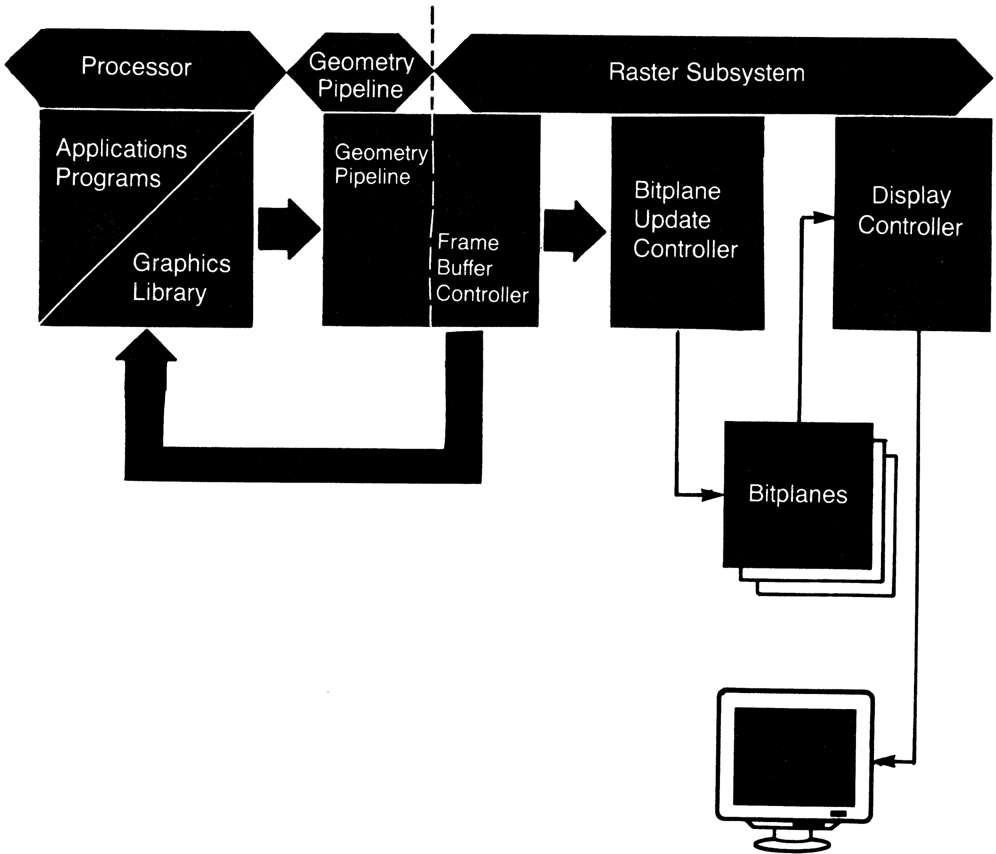
- the raster subsystem is no longer updated
- positions of transformed points are directed from the display and rerouted for use with the application.

For example, if you call `pnt` in feedback mode, the Geometry Pipeline performs its normal function of transforming, clipping, and scaling the world coordinates to screen coordinates. Then the system stores the command code for `pnt` along with the new transformed coordinates in the feedback buffer. (If the point is outside the clipping region, nothing is added to the buffer.) The following sections discuss the functions of the Geometry Pipeline, how to start and stop feedback mode, how to pass markers through the Geometry Pipeline into the feedback buffer, and how to interpret the contents of the buffer.

Note: Feedback may not be compatible with future products.

10.1 The Geometry Pipeline

To display images on the screen, the applications/graphics processor sends graphics commands through the Geometry Pipeline to the raster display subsystem. (See Figure 10-1.) All the commands the processor calls constitute two classes:



In feedback mode, the output of the Geometry Pipeline is returned to the applications/graphics processor.

Figure 10-1. Feedback

- commands that the Geometry Pipeline interprets, which include the move, draw, point, polygon, curve, and matrix manipulation commands
- commands that the Geometry Pipeline doesn't change, which include the color, character string, pattern, linestyle, and display mode commands

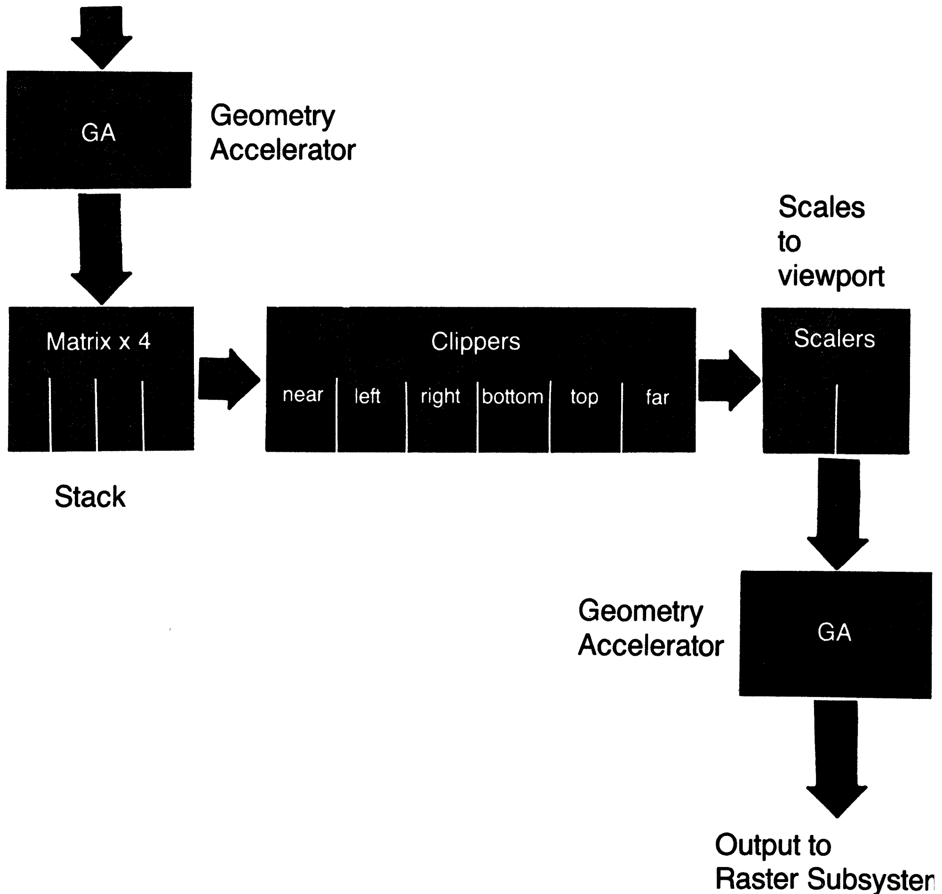
The Geometry Pipeline consists of two geometry accelerators and 10 or 12 Geometry Engines. When the pipeline is initialized, each Geometry Engine is configured to perform a specific function. (See Figure 10-2.) There are three Geometry Pipeline subsystems:

- The *matrix subsystem* consists of four Geometry Engines configured as matrix multipliers, one for each column in a 4x4 matrix.
- The *clipping subsystem* consists of four or six Geometry Engines configured as clippers, one for each clipping plane of the drawing space.
- The *scaling subsystem* consists of two scalers that scale coordinates to the viewport.

Systems with ten Geometry Engines have only four clippers and do not clip to the near and far clipping planes.

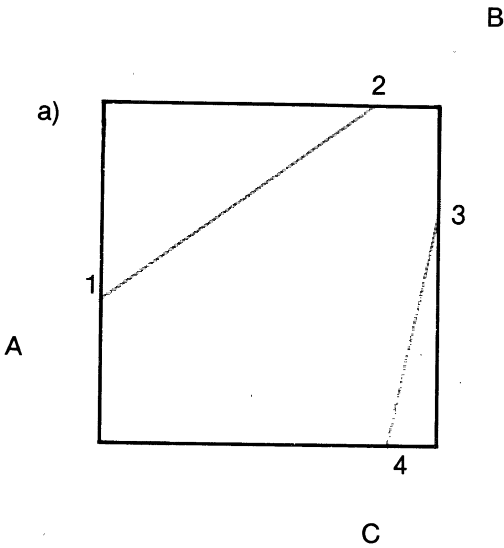
The applications/graphics processor issues graphics commands in user-defined coordinates. The first geometry accelerator converts the user-defined coordinates to floating point numbers (if necessary) and adds zeros and ones (1s) to make four-dimensional homogeneous coordinates. The first four Geometry Engines—the matrix multipliers—transform the coordinates to a normalized screen space. Each engine performs the calculations for one column of the *current transformation matrix*. The IRIS computes the current transformation matrix from the modeling, viewing, and projection transformation commands you issue, and then automatically loads it into the Geometry Engines. You can also directly define the transformation matrix in the Geometry Engines with the matrix manipulation commands (e.g., `loadmatrix`) discussed in Chapter 4, Coordinate Transformations, Section 4.5.

The next four Geometry Engines—the clippers—clip the normalized coordinates to the top, bottom, left, and right clipping planes. Two optional Geometry Engines clip to the near and far clipping planes. The clipping process can clip out entire commands or it can cause new commands to be generated. Figure 10-3(a) shows how clipping can cause a new `move` to be inserted in the command stream. Figure 10-3(b) shows how a three-point polygon turns into a seven-point polygon after passing through the clippers.



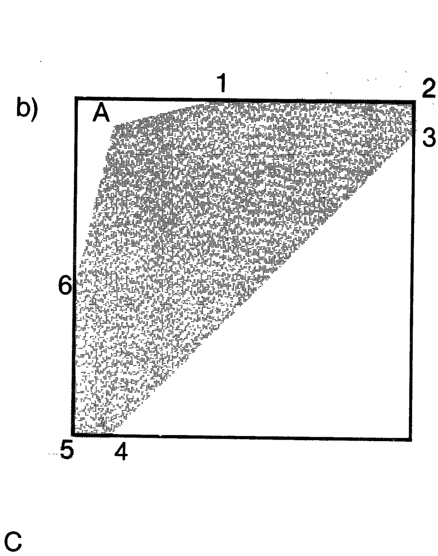
The Geometry Pipeline consists of two geometry accelerators and three Geometry Engine subsystems: the matrix multipliers, the clippers, and the scalers.

Figure 10-2. Geometry Pipeline



The sequence: move(<A>)
draw(<B>)
draw(<C>)

becomes: move(<1>)
draw(<2>)
move(<3>)
draw(<4>)



The sequence: pmv (<A>)
pdr (<B>)
pdr (<C>)
pclos

becomes: pmv (<A>)
pdr (<1>)
pdr (<2>)
pdr (<3>)
pdr (<4>)
pdr (<5>)
pdr (<6>)
pclos

The clipping subsystem of the Geometry Pipeline can generate new routines; (a) shows a new *move* inserted in the routine stream while (b) shows a three-point polygon that turns into a seven-point polygon.

Figure 10-3. The Effect of Clipping on Feedback

The final two Geometry Engines—the scalers—map the normalized, clipped coordinates to a particular area of the screen. `viewport` or `winopen` defines this area of the screen. (See *Using mex, the IRIS Window Manager*, Chapter 2 for a discussion of `winopen`; Chapter 4, Coordinate Transformations, Section 4.4, for a discussion of viewports; and Appendix B for further discussion of Geometry Engine computations.)

10.2 Feedback Mode

The IRIS feedback facility allows you to perform matrix operations and geometric computing with the Geometry Pipeline. In feedback mode, the IRIS saves the output of the Geometry Pipeline (a series of 16-bit integers) in a buffer. Each command appears as a token, followed by data values. Table 10-1 shows the tokens for commands that the Geometry Pipeline transforms, along with the data associated with each command. Note that circles, arcs, polygons, curves, and patches all appear in the feedback buffer as `move` and `draw` routines. Most attribute and viewport commands appear in the feedback buffer. Do not use them in feedback mode because the system does not transform them and this complicates the interpretation of the buffer. `passthrough` and `xfpt` at the bottom of the table are discussed later in this section.

Command	Normal Mode	Depth-cue Mode Z-buffer Mode
move	16,x,y	16,x,y,x',z
draw	17,x,y	17,x,y,x',z
pnt	18,x,y	18,x,y,x',z
pmv	48,x,y	48,x,y,x',z
pdr	49,x,y	49,x,y,x',z
pclos	51,x,y	51,x,y,x',z
passthrough	8,value	8,value
xfpt	56,x,y,z,w	56,x,y,z,w

Table 10-1. Command Tokens and Associated Data

feedback

`feedback` puts the IRIS in feedback mode. Its first argument, *buffer*, is the name of the buffer that saves the Geometry Pipeline output. The second argument, *size*, specifies the numbers of items the system saves. If more values than the number specified in *size* appear, the extra values are lost.

```
feedback(buffer,size)
short buffer[];
long size;

subroutine feedba(buffer,size)
integer*2 buffer(*)
integer*4 size

procedure feedback(var buffer: Short; size: longint);
```

endfeedback

`endfeedback` returns the number of values stored in the buffer.

```
long endfeedback(buffer)
short buffer[];

integer*4 function endfee(buffer)
integer*2 buffer(*)

function endfeedback(var buffer: Short): longint;
```

passthrough

`passthrough` stores the command token 8 and an associated 16-bit integer in the feedback buffer. The passed-through value acts as a marker in the feedback buffer, which you can use to match the input to the pipeline with the output from the pipeline. `passthrough` is especially useful for identifying clipped commands that do not appear in the feedback buffer.

```
passthrough(token)
short token;

subroutine passth(token)
integer*4 token

procedure passthrough(token: longint);
```

The following program issues three pnt commands in feedback mode and prints out the values stored in the feedback buffer. For the purposes of this example, the projection transformation is the default, ortho2, which maps the user-defined coordinates to screen coordinates without changing them.

■ C Program: FEEDBACK

```
#include "gl.h"

main()
{
    unsigned short buf[200];
    register i,j,num;
    float *bufptr;

    ginit();
    feedback(buf, 200); /* turns on feedback mode,
           buf is the name of the feedback buffer
           and there is room for 200 16-bit values */
    pnt(1.0,2.0,-0.2);
    passthrough(1);   /* store a marker in buf called "1" */
    pnt2(23.0,6.0);
    passthrough(2);   /* store a marker called "2" */
    pnt2i(0x123,0x234); /* arguments are in hexadecimal */
    passthrough(3);   /* store a marker called "3" */
    num = endfeedback(buf);
        /* the following section of the program prints out
        the contents of the feedback buffer */
    for (i = 0; i < num; i++) {
        if (i % 8 == 0)
            printf("\n");
        printf(" %0.4x\t", buf[i]);
    }
    printf("\n");
    printf("\n");

    gexit();
}
```

■ FORTRAN Program: FEEDBACK

```
      #INCLUDE /usr/include/fgl.h
      #INCLUDE /usr/include/fdevice.h

       integer*2 buf(200)
       integer*4 i,j,num

       call ginit
       call feedba(buf,200)
       call pnt(1.0,2.0,-0.2)
       call passth(1)
       call pnt2(23.0,6.0)
       call passth(2)
       call pnt2i($123,$234)
       call passth(3)
       num = endfee(buf)
       do 10 i = 1 , num
         write(*,111)buf(i)
    if (mod(i,8) .eq. 0) then
        print *
         end if
111      format(Z4)
10     continue
       print *
       print *
       call gexit
       stop
       end
```

The output of the program is the contents of `buf` (in hexadecimal):

```
0012    0001    0002    0008    0001    0012    0017    0006
0008    0002    0012    0123    0234    0008    0003
```

or in decimal:

```
18 1 2 8 1
18 23 6 8 2
18 291 564
```

To compute and save *z* coordinates in the feedback buffer, use z-buffer mode or depth-cue mode. (See Chapter 12 and Chapter 13.) In these modes, the Geometry Pipeline outputs five values for each command (see Table 10-1):

- a command code
- an *x* coordinate
- a *y* coordinate
- a second *x* coordinate (a dummy value, reserved for future use)
- a *z* coordinate

The following program issues three `pnt` commands with both feedback mode and depth-cue mode turned on, and prints out the contents of the feedback buffer.

■ C Program: FEEDBACK AND DEPTHCUE

```
#include "gl.h"

main()
{
    unsigned short buf[200];
    register i,j,num;
    float *bufptr;

    ginit();
    setdepth(0,10); /* set the near clipping plane to 0 and the
            far clipping plane to 10 */
    depthcue(1);  /* turn on depth-cue mode */
    feedback(buf,200);
    pnt(1.0,2.0,1.0);
    passthrough(1);
    pnt(1.0,2.0,-1.0);
    passthrough(2);
    pnt2i(0x123,0x234); /* arguments are in hexadecimal */
    passthrough(3);
    num = endfeedback(buf);
    depthcue(0);  /* turn off depth-cue mode */
        /* this section prints out the contents of buf */
    for (i = 0; i < num; i++) {
```

```
        if (i % 8 == 0)
            printf("\n");
        printf(" %0.4x\t", buf[i]);
    }
    printf("\n");
    printf("\n");

    gexit();
}
```

■ FORTRAN Program: FEEDBACK AND DEPTHCUE

```
#INCLUDE /usr/include/fgl.h

      integer*2 buf(200)
      integer*4 i,j,num

      call ginit
      call setdep(0,10)
      call depthc(1)

      call feedba(buf,200)
      call pnt(1.0,2.0,1.0)
      call passth(1)
      call pnt(1.0,2.0,-1.0)
      call passth(2)
      call pnt2i($123,$234)
      call passth(3)
      num = endfee(buf)
      do 10 i = 1 , num
        write(*,111)buf(i)
    if (mod(i,8).eq. 0) then
       print *
        end if
```

```
111     format(Z6$)
10    continue
      print *
      print *
      call gexit
      stop
      end
```

The output of the program is:

```
0012    0001    0002    0001    0000    0008    0001    0012
0001    0002    0001    000b    0008    0002    0012    0123
0234    0123    0005    0008    0003
```

This can be translated into:

```
18 1 2 1 0
8 1
18 1 2 1 11
8 2
18 291 564 291 5
```

The first number in each list is a command code. For the `pnt` commands, the four values following the first number are x, y, x', and z .

xfpt

All previous examples involve feedback from standard Graphics Library routines. By the time data reaches the feedback buffer, all the coordinates have been converted to screen space pixel coordinates. `xfpt` (transform point) is a special Graphics Library routine that multiplies an input vector by the top matrix on the matrix stack, and passes the unclipped, unscaled values through to the output buffer. The input vector consists of four floating point values. The output in the feedback buffer is the xfpt token (56), followed by four floating point values. Call `xfpt` only in feedback mode.

Note: `xfpt` does not speed execution in the 3000 series or the 2400T, which have floating point boards. It is more efficient to write ordinary matrix multiplication subroutines.

`xfpt` may not be compatible with future products.

```
xfpt(x, y, z)
Coord x, y, z;

subroutine xfpt(x, y, z)
real x, y, z

procedure xfpt(x, y, z: Coord);
```

The following program demonstrates the `xfpt` routine. Several points are sent through feedback and multiplied with the identity matrix. The 16-bit buffer is parsed and read as 32-bit floating point values. `xfpt` requires the first three values in the four-component vector. The fourth component, *w*, is assumed to be 1.0. There are a variety of `xfpt` routines that specify two, three, or four components in the vector: `xfpt2` requires the x and y values and assumes z=0.0 and w=1.0. `xfpt4` requires all four components.

■ C Program: xfpt

```
#include "gl.h"
    /* define the matrix that will transform the points
        (this particular matrix does not change the points) */
float idmat[4][4] = {
    1.0, 0.0, 0.0, 0.0,
    0.0, 1.0, 0.0, 0.0,
    0.0, 0.0, 1.0, 0.0,
    0.0, 0.0, 0.0, 1.0};

main()
{
    unsigned short buf[200];
    register i,j,num;
    float *bufptr;

    ginit();
    pushmatrix(); /* save the current transformation matrix */
    loadmatrix(idmat); /* load the transformation matrix */
    feedback(buf,50);
    xfpt(1.0,2.0,3.0);
    xfpt4(5.0,6.0,7.0,3.14159);
    xfpt2i(7,8);
    num = endfeedback(buf);
    popmatrix(); /* restore original transformation matrix */
```

```
        /* print out buf */
    for (i = 0; i < num; i++) {
        if i % 8 == 0)
            printf("\n");
        printf(" %0.4x\t", buf[i]);
    }
    printf("\n");
    printf("\n");
        /* print out floating point versions of the coordinates */
    for (i = 0; i < 3; i++) {
        bufptr = (float *)(&buf[1 + i*9]);
        for (j = 0; j < 4; j++)
            printf(" %f\t", *bufptr++);
        printf("\n");
    }
    gexit();
}
```

■ FORTRAN Program: xfpt

```
#INCLUDE /usr/include/fgl.h
#INCLUDE /usr/include/fdevice.h

      integer*2 buf(200)
      real rbuf(100),rbuf2(100)
      integer i,j,k,inum
      real idmat(4,4)
      real postra(15,4)
      equivalence (rbuf(1),buf(1))
      equivalence (rbuf2(1),buf(2))

      data idmat/1.0,0.0,0.0,0.0,
     +           0.0,1.0,0.0,0.0,
     +           0.0,0.0,1.0,0.0,
     +           0.0,0.0,0.0,1.0/
      call ginit
      call pushma
      call loadma(idmat)
      call feedba(buf,200)
```

```
      j=9
      do 10 i=1,inum
        print 101, buf(i)
101          format(Z6$)
        j=j-1
        if (j .eq. 0) then
           j = 9
           print *
            end if
10    continue
      print *

      k =1
      do 20 i = 1, inum,9
        if (mod(i,2) .eq. 0) then
          do 30 j= 1,4
            postra(k,j) = rbuf(i/2+j)
30             continue
        else
          do 40 j = 1, 4
            postra(k,j) = rbuf2(i/2+j)
40             continue
        end if
        k = k+ 1
20    continue

      i = 4
      do 50 j =1 , 4
    do 60 k = 1,3
        print 102, postra(k,j)
102          format(F10.6$)
60        continue
    print *
50    continue
      print *
```

```
print *
call gexit
stop
end
```

The output of the program is:

```
0038    3f80    0000    4000    0000    4040    0000    3f80
0000    0038    40a0    0000    40c0    0000    40e0    0000
4049    0fd0    0038    40e0    0000    4100    0000    0000
0000    3f80    0000

1.000000        2.000000        3.000000        1.000000
5.000000        6.000000        7.000000        3.141590
7.000000        8.000000        0.000000        1.000000
```

Since the identity matrix was used, the coordinates in the feedback buffer are equal to the original coordinates.

11. Curves and Surfaces

The IRIS Graphics Library provides routines that represent curved lines and surfaces. The first section in this chapter describes the mathematical basis for the curve facility. The second and third sections describe how to draw curves and surfaces.

A curve segment is drawn by specifying:

- a set of four control points
- a basis, which defines how the system uses the control points to determine the shape of the segment

Complex curved lines can be created by joining several curve segments end to end. The curve facility provides the means for making smooth joints between the segments.

Three-dimensional surfaces, or *patches*, are represented by a 'wireframe' of curve segments. A patch is drawn by specifying:

- a set of 16 control points
- the number of curve segments to be drawn in each direction of the patch
- the two bases that define how the control points determine the shape of the patch

Complex surfaces can be created by joining several patches into one large patch.

11.1 Curve Mathematics

The mathematical basis for the IRIS curve facility is the *parametric cubic curve*. The curves in most applications are too complex to be represented by a single curve segment and instead must be represented by a series of curve segments joined end to end. To create smooth joints, you must control the positions and curvatures at the endpoints of curve segments. Parametric cubic curves are the lowest-order representation of curve segments that provide continuity of position, slope, and curvature at the point where two curve segments meet.

A parametric cubic curve has the property that x, y, and z can be defined as third-order polynomials for variable t:

$$x(t) = a_x t^3 + b_x t^2 + c_x t + d_x$$

$$y(t) = a_y t^3 + b_y t^2 + c_y t + d_y$$

$$z(t) = a_z t^3 + b_z t^2 + c_z t + d_z$$

A cubic curve segment is defined over a range of values for t (usually $0 \le t \le 1$), and can be expressed as a vector product:

$$C(t) = at^3 + bt^2 + ct + d$$

$$= \begin{bmatrix} t^3 t^2 t 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$$

$$= TM$$

The IRIS approximates the shape of a curve segment with a series of straight line segments. The endpoints for all the line segments can be computed by evaluating the vector product *C(t)* for a series of t values between 0 and 1. The shape of the curve segment is determined by the coefficients of the vector product, which are stored in column vector M. These coefficients can be expressed as a function of a set of four control points. Thus, the vector product becomes

$$C(t) = TM = T(BG)$$

where G is a set of four control points, or the *geometry*, and B is a matrix called the *basis*. The basis matrix is determined from a set of constraints that express how the shape of the curve segment relates to the control points. For example, a constraint might be that one endpoint of the curve segment is located at the first control point; or the tangent vector at that endpoint lies on the line segment formed by the first two control points. When the vector product C is solved for a particular set of constraints, the coefficients of the vector product are identified as a function of four variables (the control points). Then, given four control points, the vector product can be used to generate the points on the curve segment.

Three classes of cubic curves are discussed here: Bezier, Cardinal spline, and B-spline. Other classes of cubic curves are described in a paper by J.H. Clark.[1] The set of constraints that define each class is given, along with the basis matrix derived from those constraints. Section 11.2 tells how to use the basis matrices to draw curve segments.

11.1.1 Bezier Cubic Curve

A *Bezier* cubic curve segment passes through the first and fourth control points and uses the second and third points to determine the shape of the curve segment. Of the three kinds of curves, the Bezier form provides the most intuitive control over the shape of the curve. The Bezier basis matrix is derived from the following four constraints:

- One endpoint of the segment is located at p_1:

  $Bezier(0) = p_1$

- The other endpoint is located at p_4:

  $Bezier(1) = p_4$

---

1. Clark, James H., 'Parametric Curves, Surfaces and Volumes in Computer Graphics and Computer-Aided Geometric Design,' *Computer Systems Laboratory, Technical Report No. 221*, Stanford University, November 1981.

- The first derivative, or slope, of the segment at one endpoint is equal to this value:

  $Bezier(0)' = 3(p_2 - p_1)$

- The first derivative at the other endpoint is equal to this value:

  $Bezier(1)' = 3(p_4 - p_3)$

Solving for these constraints yields the following equation:

$$Bezier(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix}$$

$$= TM_bG_b$$

You can generate all the points on the Bezier cubic curve segment from p_1 to p_4 by evaluating *Bezier(t)* for $0 \le t \le 1$. (It is more efficient, however, to construct a forward difference matrix that generates the points in a curve segment incrementally. Forward difference matrices are discussed in Section 11.2.)

Figure 11-1 shows three Bezier curve segments. The first segment uses points 0, 1, 2, and 3 as control points. The second uses 1, 2, 3, and 4. The third uses 2, 3, 4, and 5. You can use the technique of overlapping sets of control points more effectively with the following two classes of cubic curves to create a single large curve from a series of curve segments.

11.1.2 Cardinal Spline Cubic Curve

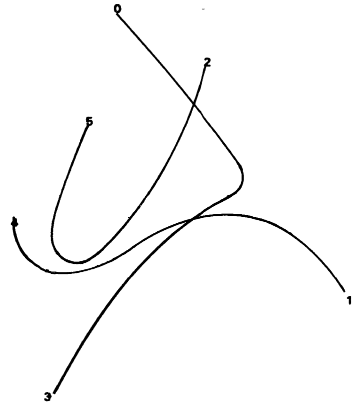
A *Cardinal spline* curve segment passes through the two interior control points and is continuous in the first derivative at the points where segments meet. The curve segment starts at p_2 and ends at p_3, and uses p_1 and p_4 to define the shape of the curve. The mathematical derivation of the Cardinal spline basis matrix can be found in Clark's paper.[2]

---

2. *Ibid.*

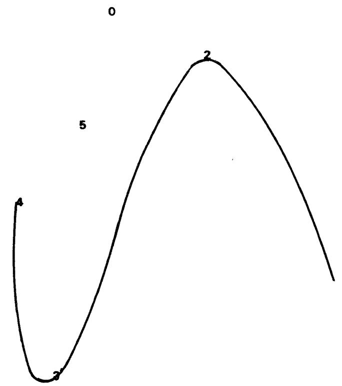
(a) Bezier

$$\begin{bmatrix} -1.0 & 3.0 & -3.0 & 1.0 \\ 3.0 & -6.0 & 3.0 & 0.0 \\ -3.0 & 3.0 & 0.0 & 0.0 \\ 1.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$



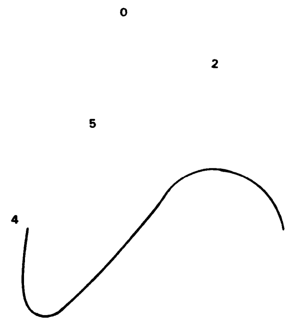
(b) Cardinal Spline

$$\begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1.0 & -2.5 & 2.0 & -0.5 \\ -0.5 & 0.0 & 0.5 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \end{bmatrix}$$



(c) B-Spline

$$\frac{1}{6}\begin{bmatrix} -1.0 & 3.0 & -3.0 & 1.0 \\ 3.0 & -6.0 & 3.0 & 0.0 \\ -3.0 & 0.0 & 3.0 & 0.0 \\ 1.0 & 4.0 & 1.0 & 0.0 \end{bmatrix}$$



Three different curves are shown with appropriate basis matrices. With the Bezier basis matrix, three sets of overlapping control points result in three separate curve segments. With the Cardinal spline and B-spline matrices, the same overlapping sets of control points result in three joined curve segments.

Figure 11-1. Bezier, Cardinal, and B-Spline Curves

The Cardinal spline basis matrix is derived from the following four constraints:

$Cardinal(0) = p_2$

$Cardinal(1) = p_3$

$Cardinal(0)' = a(p_3 - p_1)$

$Cardinal(1)' = a(p_4 - p_2)$

The scalar coefficient a must be positive; it determines the length of the tangent vector at point p_2 : $tangent_2 = a(p_3 - p_1)$ and p_3 : $tangent_3 = a(p_4 - p_2)$. Solving for these constraints yields the following equation:

$$Cardinal(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -a & 2-a & -2+a & a \\ 2a & -3+a & 3-2a & -a \\ -a & 0 & a & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix}$$

$$= TM_cG_c$$

The three joined Cardinal spline curve segments in Figure 11-1 use the same three sets of control points as the Bezier curve segments. Many different bases have Cardinal spline properties. You can derive the different bases by trying different values of a.

11.1.3 B-Spline Cubic Curve

In general, a *B-spline* curve segment does not pass through any control points, but is continuous in both the first and second derivatives at the points where segments meet. Thus, a series of joined B-spline curve segments is smoother than a series of Cardinal spline segments. (See Figure 11-1.)

The B-spline basis matrix is derived from the following four constraints:

$$B\text{-}spline(0)' = \frac{(p_3 - p_1)}{2}$$

$$B\text{-}spline(1)' = \frac{(p_4 - p_2)}{2}$$

$$B\text{-}spline(0)'' = p_1 - 2p_2 + p_3$$

$$B\text{-}spline(1)'' = p_2 - 2p_3 + p_4$$

Solving for these constraints yields the following equation:

$$B\text{-}spline(t) = \begin{bmatrix} t^3 t^2 t 1 \end{bmatrix} \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix}$$

$$= TM_B G_B$$

11.2 Drawing Curves

Drawing a curve segment on the screen involves four steps:

1. Define and name a basis matrix with `defbasis`.
2. Select a defined basis matrix as the *current basis matrix* with `curvebasis`.
3. Specify the number of line segments used to approximate each curve segment with `curveprecision`.
4. Draw the curve segment using the current basis matrix, the current curve precision, and the four control points with `crv`. `rcrv` draws a rational curve.

defbasis

`defbasis` defines and names a basis matrix to generate curves and patches. *mat* is saved and is associated with *id*. Use *id* in subsequent calls to `curvebasis` and `patchbasis`.

```
defbasis(id, mat)
long id;
Matrix mat;

subroutine defbas(id, mat)
integer*4 id
real mat(4,4)

procedure defbasis(id: Short; var mat: Matrix);
```

curvebasis

`curvebasis` selects a basis matrix (defined by `defbasis`) as the *current basis matrix* to draw curve segments.

```
curvebasis(basisid)
short basisid;

subroutine curveb(basisid)
integer*4 basisid

procedure curvebasis(basisid: Short);
```

curveprecision

`curveprecision` specifies the number of line segments used to draw a curve. Whenever `crv`, `crvn`, `rcrv`, or `rcrvn` executes, a number of straight line segments (*nsegments*) approximates each curve segment. The greater the value of *nsegments*, the smoother the curve, but the longer the drawing time.

```
curveprecision(nsegments)
short nsegments;

subroutine curvep(nsegments)
integer*4 nsegments

procedure curveprecision(nsegments: longint);
```

crv

`crv` draws the curve segment using the current basis matrix, the current curve precision, and the four control points specified in the argument to `crv`.

```
crv(geom)
Coord geom[4][3];

subroutine crv(geom)
real geom(3,4)

procedure crv(var geom: Coord);
```

When you issue `crv`, a matrix is built from the geometry, the current basis, and the current precision:

$$M = F_{precision} M_{basis} G_{geom}$$

$$= \begin{bmatrix} \frac{6}{n^3} & 0 & 0 & 0 \\ \frac{6}{n^3} & \frac{2}{n^2} & 0 & 0 \\ \frac{1}{n^3} & \frac{1}{n^2} & \frac{1}{n} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} M_{basis} G_{geom}$$

where n = the current precision. The bottom row of the resulting transformation matrix identifies the first of n points that describe the curve. To generate the remaining points in the curve, the following algorithm is used to iterate the matrix as a forward difference matrix. The third row is added to the fourth row, the second row is added to the third row, and the first row is added to the second row. The fourth row is then output as one of the points on the curve.

```
/* This is the forward difference algorithm */
/* M is the current transformation matrix */
move (M[3][0]/M[3][3], M[3][1]/M[3][3], M[3][2]/M[3][3]);
/* iteration loop */
for (cnt = 0; cnt < iterationcount; cnt++) {
   for (i=3; i>0; i--)
           for (j=0; j<4; j++)
                  M[i][j] = M[i][j] + M[i-1][j];
   draw(M[3][0]/M[3][3], M[3][1]/M[3][3], M[3][2]/M[3][3]);
}
```

Each iteration draws one line segment of the curve segment. Note that if the precision matrix on the previous page is iterated as a forward difference matrix it generates the sequence of points:

$$(0, 0, 0, 1); \; ((\frac{1}{n})^3, (\frac{1}{n})^2, \frac{1}{n}, 1); \; ((\frac{2}{n})^3, (\frac{2}{n})^2, \frac{2}{n}, 1); \; ((\frac{3}{n})^3, (\frac{3}{n})^2, \frac{3}{n}, 1); \; \ldots.$$

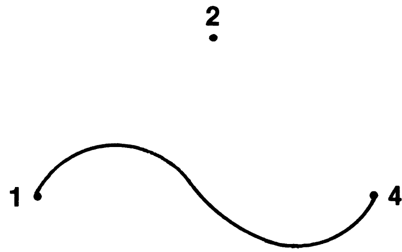
This is the same sequence of points generated by

$$t = 0, \frac{1}{n}, \frac{2}{n}, \frac{3}{n}, \ldots$$

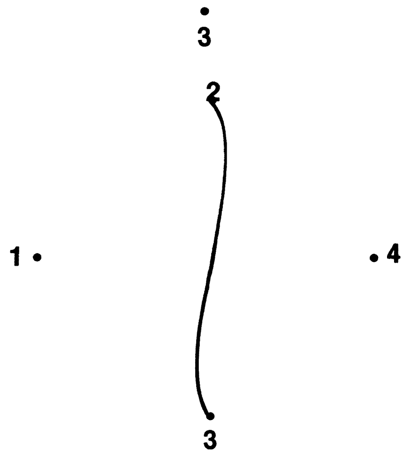
for the vector
$(t^3, t^2, t, 1)$.

The following program draws the three curve segments in Figure 11-2. All use the same set of four control points, which is contained in *geom1*. The three basis matrix arrays (*beziermatrix, cardinalmatrix,* and *bsplinematrix*) contain the values discussed in the previous section. Before `crv` (or `rcrv`) is called, a basis and precision matrix must be defined. This is also true if the routines are compiled into an object.

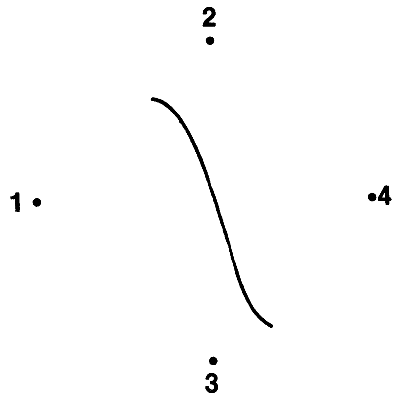
Bezier



Cardinal spline



B-spline



Each of the above curve segments uses the same set of four control points and the same precision, but a different basis matrix.

Figure 11-2. Curve Segments

■ C Program: CURVE SEGMENTS

```
#include "gl.h"

Matrix beziermatrix = {
    { -1, 3, -3, 1 },
    { 3, -6, 3, 0 },
    { -3, 3, 0, 0 },
    { 1, 0, 0, 0 }
    };
Matrix cardinalmatrix = {
    { -0.5, 1.5, -1.5, 0.5 },
    { 1.0, -2.5, 2.0, -0.5 },
    { -0.5, 0, 0.5, 0 },
    { 0, 1, 0, 0 }
    };
Matrix bsplinematrix = {
    { -1.0/6.0, 3.0/6.0, -3.0/6.0, 1.0/6.0 },
    { 3.0/6.0, -6.0/6.0, 3.0/6.0, 0 },
    { -3.0/6.0, 0, 3.0/6.0, 0 },
    { 1.0/6.0, 4.0/6.0, 1.0/6.0, 0 }
    };
#define BEZIER 1
#define CARDINAL 2
#define BSPLINE 3
Coord geom1[4][3] = {
    { 100.0, 100.0, 0.0},
    { 200.0, 200.0, 0.0},
    { 200.0, 0.0, 0.0},
    { 300.0, 100.0, 0.0}
    };

main ()
{
ginit();
color(BLACK);
clear();
defbasis(BEZIER,beziermatrix); /* define a basis matrix
        called BEZIER */
curvebasis(BEZIER); /* identify the BEZIER matrix
        as the current basis matrix */
curveprecision(20); /* set the current precision to 20
```

```
        (the curve segment will be drawn using 20 line segments) */
color(RED);
crv(geom1); /* draw the curve based on the four control
           points in geom1 */

defbasis(CARDINAL,cardinalmatrix); /* a new basis is defined */
curvebasis(CARDINAL); /* the current basis is reset */
        /* note that the curveprecision does not have to
        be restated unless it is to be changed */
color(BLUE);
crv(geom1); /* a new curve segment is drawn */

defbasis(BSPLINE,bsplinematrix); /* a new basis is defined */
curvebasis(BSPLINE); /* the current basis is reset */
color(GREEN);
crv(geom1); /* a new curve segment is drawn */

gexit();
}
```

■ FORTRAN Program: CURVE SEGMENTS

```
     #INCLUDE 'fgl.h'
C
      REAL BEZMAT(4,4), CARMAT(4,4), BSPMAT(4,4), GEOM1(3,4)
      INTEGER*2 BEZI, CARD, BSPL
      REAL ASIXTH, MSIXTH, TTHRDS
      INTEGER I, NOP
      PARAMETER ( BEZIER = 1 )
      PARAMETER ( CARDIN = 2 )
      PARAMETER ( BSPLIN = 3 )
      PARAMETER ( ASIXTH = 1.0/6.0 )
      PARAMETER ( MSIXTH = -1.0/6.0 )
      PARAMETER ( TTHRDS = 2.0/3.0 )
C
      DATA BEZMAT /-1.0,  3.0, -3.0,  1.0,
     2              3.0, -6.0,  3.0,  0.0,
     3             -3.0,  3.0,  0.0,  0.0,
     4              1.0,  0.0,  0.0,  0.0/
      DATA CARMAT /-0.5,  1.5, -1.5,  0.5,
```

```
     2              1.0, -2.5,  2.0, -0.5,
     3             -0.5,  0.0,  0.5,  0.0,
     4              0.0,  1.0,  0.0,  0.0/
      DATA BSPMAT /MSIXTH,    0.5,   -0.5, ASIXTH,
     2                0.5,   -1.0,    0.5,    0.0,
     3               -0.5,    0.0,    0.5,    0.0,
     4             ASIXTH, TTHRDS, ASIXTH,    0.0/
      DATA GEOM1 /100.0, 100.0, 0.0,
     2            200.0, 200.0, 0.0,
     3            200.0,   0.0, 0.0,
     4            300.0, 100.0, 0.0/
C
      CALL GINIT
      CALL COLOR(BLACK)
      CALL CLEAR()
C       define a basis matrix called BEZIER
      CALL DEFBAS(BEZIER,BEZMAT)
C       identify the BEZIER matrix as the current basis matrix
      CALL CURVEB(BEZIER)
C       set the current precision to 20
C       (the curve segment will be drawn using 20 line segments)
      CALL CURVEP(20)
      CALL COLOR(RED)
C       draw the curve based on the four control points in geom1
      CALL CRV(GEOM1)
C
C       define a new basis
      CALL DEFBAS(CARDIN,CARMAT)
C       reset the current basis
      CALL CURVEB(CARDIN)
C       note that the curveprecision does not have to
C       be restated unless it is to be changed
      CALL COLOR(BLUE)
C       draw a new curve segment
      CALL CRV(GEOM1)
C
C       define a new basis
      CALL DEFBAS(BSPLIN,BSPMAT)
C       reset the current basis
      CALL CURVEB(BSPLIN)
      CALL COLOR(GREEN)
```

```
C        draw a new curve segment
      CALL CRV(GEOM1)
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C       Installation note:
C        This is a delay loop, which may require adjustment.
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      CALL GFLUSH()
      DO 1000 I=1,1000000
      NOP = I * 2
1000  CONTINUE
      CALL GEXIT()
      STOP
      END
```

crvn

`crvn` takes a series of `n` control points and draws a series of cubic spline or rational cubic spline curve segments using the current basis and precision; `rcrvn` draws rational splines. The control points specified in *geom* determine the shapes of the curve segments and are used four at a time. If the current basis is a B-spline, Cardinal spline, or basis with similar properties, the curve segments are joined end to end and appear as a single curve. Calling `crvn` has the same effect as calling a sequence of `crv` with overlapping control points (see Figure 11-1).

```
crvn(n, geom)
long n;
Coord geom[][3];

subroutine crvn(n,geom)
integer*4 n
real geom(3,n)

procedure crvn(n: longint; var geom: Coord);
```

When you issue `crvn` with a Cardinal spline or B-spline basis, it produces a single curve. However, `crvn` issued with a Bezier basis produces several separate curve segments.

As with `crv` and `rcrv`, a precision and basis must be defined before calling `crvn` or `rcrvn`. This is true even if the routines are compiled into objects.

The following program draws the three joined curve segments in Figure 11-1 using `crvn`. *geom2* contains six control points.

■ C Program: JOINED CURVE SEGMENTS

```
#include "gl.h"

Matrix beziermatrix = {
    { -1, 3, -3, 1 },
    { 3, -6, 3, 0 },
    { -3, 3, 0, 0 },
    { 1, 0, 0, 0 }
    };
Matrix cardinalmatrix = {
    { -0.5, 1.5, -1.5, 0.5 },
    { 1.0, -2.5, 2.0, -0.5 },
    { -0.5, 0, 0.5, 0 },
    { 0, 1, 0, 0 }
    };
Matrix bsplinematrix = {
    { -1.0/6.0, 3.0/6.0, -3.0/6.0, 1.0/6.0 },
    { 3.0/6.0, -6.0/6.0, 3.0/6.0, 0 },
    { -3.0/6.0, 0, 3.0/6.0, 0 },
    { 1.0/6.0, 4.0/6.0, 1.0/6.0, 0 }
    };
#define BEZIER 1
#define CARDINAL 2
#define BSPLINE 3
Coord geom2[6][3] = {
    { 150.0, 400.0, 0.0},
    { 350.0, 100.0, 0.0},
    { 200.0, 350.0, 0.0},
    { 50.0, 0.0, 0.0},
    { 0.0, 200.0, 0.0},
    { 100.0, 300.0, 0.0},
    };
main ()
{
    ginit();
    color(BLACK);
    clear();
```

```
    defbasis(BEZIER,beziermatrix); /* define a basis matrix
            called BEZIER */
    defbasis(CARDINAL,cardinalmatrix); /* a new basis is defined */
    defbasis(BSPLINE,bsplinematrix); /* a new basis is defined */

    curvebasis(BEZIER); /* the Bezier matrix becomes
                                the current basis */
    curveprecision(20); /* the precision is set to 20 */
    color(RED);
    crvn(6, geom2); /* the crvn command called with a Bezier
        basis causes three separate curve segments to be drawn */

    curvebasis(CARDINAL); /* the Cardinal basis becomes the
                                current basis */
    color(GREEN);
       crvn(6, geom2); /* the crvn command called with a Cardinal
         spline basis causes a smooth curve to be drawn */

    curvebasis(BSPLINE); /* the B-spline basis becomes the
                                current basis*/

    color(BLUE);
       crvn(6, geom2);  /* the crvn command called with a B-spline
        basis causes the smoothest curve to be drawn */

    gexit();
}
```

■ FORTRAN Program: JOINED CURVE SEGMENTS

```
      #INCLUDE /usr/include/fgl.h
      #include /usr/include/fdevice.h

      real bezmat(4,4),carmat(4,4),bspmat(4,4),geom2(3,6)
      integer*2 bezi,card,bspl
      real asizth,msixth,tthrds
      integer i,nop
      parameter(bezier =1)
      parameter(cardin =2)
      parameter(bsplin =3)
      parameter(asixth =1.0/6.0)
```

```
 parameter(msixth =-1.0/6.0)
 parameter(tthrds =2.0/3.0)

 data bezmat /-1.0,3.0,-3.0,1.0,
+             3.0,-6.0,3.0,0.0,
+            -3.0,3.0,0.0,0.0,
+             1.0,0.0,0.0,0.0/

 data carmat/-0.5,1.5,-1.5,0.5,
+             1.0,-2.5,2.0,-0.5,
+            -0.5,0.0,0.5,0.0,
+             0.0,1.0,0.0,0.0/

 data bspmat /msixth, 0.5, -0.5, asixth,
+             0.5   ,-1.0,  0.5, 0.0,
+            -0.5,   0.0,   0.5, 0.0,
+            asixth, tthrds, asixth, 0.0/

 data geom2 /150.0,400.0,0.0,
+            350.0,100.0,0.0,
+            200.0,350.0,0.0,
+            50.0,0.0,0.0,
+            0.0,200.0,0.0,
+            100.0,300.0,0.0/

 call ginit
 call cursof
 call color(0)
 call clear
 call defbas(bezier, bezmat)
 call defbas(cardin, carmat)
 call defbas(bsplin, bspmat)
 call curveb(bezier)
 call curvep(20)
 call color(RED)
 call crvn(6,geom2)

 call curveb(cardin)

 call color(GREEN)
 call crvn(6,geom2)
```

```
      call curveb(bsplin)
      call color(BLUE)

      call crvn(6,geom2)

 999  continue
      if(.not. getbut(RIGHTM))GO TO 999
      call color(BLACK)
      call clear
      call curson
      call gexit
      stop
      end
```

curveit

The iteration loop of the forward difference algorithm is implemented in the Geometry Pipeline. `curveit` provides direct access to this facility, making it possible to generate a curve directly from a forward difference matrix. `curveit` iterates the current matrix (the one on top of the matrix stack) *iterationcount* times. Each iteration draws one of the line segments that approximate the curve. `curveit` does not execute the initial `move` in the forward difference algorithm. A `move(0.0,0.0,0.0)` must precede `curveit` so that the correct first point is generated from the forward difference matrix.

```
curveit(iterationcount)
short iterationcount;

subroutine curvei(count)
integer*4 count

procedure curveit(iterationcount: longint);
```

The following program draws the Bezier curve segment in Figure 11-2 using `curveit`. The Cardinal spline and B-spline curve segments could be drawn using a similar sequence of commands—only the basis matrix would be different.

■ C Program: BEZIER CURVE

```
#include "gl.h"

Matrix beziermatrix = {
    { -1, 3, -3, 1 },
    { 3, -6, 3, 0 },
    { -3, 3, 0, 0 },
    { 1, 0, 0, 0 }
    };
#define BEZIER 1
Matrix geom1[4][3] = {
    { 100.0, 100.0, 0.0, 1.0},
    { 200.0, 200.0, 0.0, 1.0},
    { 200.0, 0.0, 0.0, 1.0},
    { 300.0, 100.0, 0.0, 1.0}
    };
Matrix precisionmatrix = {
    { 6.0/8000.0, 0, 0, 0},
    { 6.0/8000.0, 2.0/400.0, 0, 0},
    { 1.0/8000.0, 1.0/400.0, 1/20.0, 0},
    { 0, 0, 0, 1}
    };
main ()
{
    ginit();
    color(BLACK);
    clear();
    pushmatrix(); /* the current transformation matrix on the
        matrix stack is saved */
    multmatrix(geom1); /* the product of the current transformatior
        matrix and the matrix containing the control points becomes
        the new current transformation matrix */
    multmatrix(beziermatrix); /* the product of the basis matrix
        and the current transformation matrix becomes the new
        current transformation matrix */
    multmatrix(precisionmatrix); /* the product of the precision
        matrix and the current transformation matrix becomes
        the new current transformation matrix*/
    move(0.0,0.0,0.0); /* this command must be issued so that
        the correct first point is generated by the curveit
        command */
```

```
    color(RED);
    curveit(20); /* a curve consisting of 20 line segments
            is drawn */
    popmatrix(); /* the original transformation matrix is
        restored */
    gexit();
}
```

11.2.1 Rational Curves

Cubic splines have been the focus of discussion. Cubic splines are splines whose x, y, and z coordinates can be expressed as a cubic polynomial in t.

The IRIS graphics hardware actually works in homogeneous coordinates x, y, z, and w, where 3-D coordinates are given by x/w, y/w, and z/w. w is normally the constant 1, so the homogeneous character of the system is hidden.

In fact, the w coordinate can also be expressed as a cubic function of t, so that the 3-D coordinates of points along the curve are given as a quotient of two cubic polynomials. The only constraint is that the denominator for all three coordinates must be the same. When w is not the constant 1, but some cubic polynomial function of t, the curves generated are usually called rational cubic splines.

A circle is a useful example. There is no cubic spline that exactly matches any short segment of a circle, but if x, y, z, and w are defined as:

$$x(t) = t^2-1$$
$$y(t) = 2t$$
$$z(t) = 0$$
$$w(t) = t^2+1$$

the real coordinates

$$(x/w, y/w, z/w) = \left(\frac{t^2-1}{t^2+1}, \frac{2t}{t^2+1}, 0\right)$$

all lie on the circle with center at *(0,0,0)* in the x-y plane with radius *1* (exactly). All the conic sections (ellipses, hyperbolas, parabolas) can be similarly defined.

For rational splines, the basis defintions are identical to those for cubic splines as are the precision specifications. The only difference is that the geometry matrix must be specified in four-dimensional homogeneous coordinates. This is done with `rcrv`:

```
rcrv(geom)
coord geom[4][4];
```

`rcrv` is exactly analogous to `crv` except that *w* coordinates are included in the control point definitions.

rcrv

`rcrv` draws a rational curve segment using the current basis matrix, the current curve precision, and the four control points specified in the its argument.

```
rcrv(geom)
Coord geom[4] [4];

subroutine rcrv(geom)
real geom(4,4)

procedure crv(var geom: Coord);
```

rcrvn

`rcrvn` takes a series of *n* control points and draws a series of rational cubic spline curve segments using the current basis and precision. The control points specified in *geom* determine the shapes of the curve segments and are used four at a time.

```
rcrvn(n, geom)
long n;
Coord geom[][4];

subroutine rcrvn(n,geom)
integer*4 n
real geom(4,n)

procedure rcrvn(n: longint; var geom: Coord)
```

11.3 Drawing Surfaces

The method for drawing surfaces is similar to that for drawing curves. A *surface patch* appears on the screen as a 'wireframe' of curve segments. A set of user-defined control points determines the shape of the patch. A complex surface consisting of several joined patches can be created by using overlapping sets of control points and the *B-spline* and *Cardinal spline* curve bases discussed in Section 11.1.

The mathematical basis for the IRIS surface facility is the *parametric bicubic surface*. Bicubic surfaces can provide continuity of position, slope, and curvature at the points where two patches meet. The points on a bicubic surface are defined by parametric equations for x, y, and z. The parametric equation for x is:

$$x(u,v) = a_{11}u^3v^3+a_{12}u^3v^2+a_{13}u^3v+a_{14}u^3$$

$$+a_{21}u^2v^3+a_{22}u^2v_2+a_{23}u^2v+a_{24}u^2$$

$$+a_{31}uv^3+a_{32}uv^2+a_{33}uv+a_{34}u$$

$$+a_{41}v^3+a_{42}v^2+a_{43}v+a_{44}$$

(The equations for y and z are similar.) The points on a bicubic patch are defined by varying the parameters u and v from 0 to 1. If one parameter is held constant and the other is varied from 0 and 1, the result is a cubic curve. Thus, a wireframe patch can be created by holding u constant at several values and using the IRIS curve facility to draw curve segments in one direction, and then doing the same for v in the other direction.

There are five steps involved in drawing a surface patch:

1. The appropriate curve bases are defined using `defbasis`. (See Section 11.1.) A Bezier basis provides intuitive control over the shape of the patch. The Cardinal spline and B-spline bases allow smooth joints to be created between patches.

2. A basis for each of the directions in the patch, u and v, must be specified with `patchbasis`. Note that the u basis and the v basis do not have to be the same.

3. The number of curve segments to be drawn in each direction is specified by `patchcurves`. A different number of curve segments can be drawn in each direction.

4. The precisions for the curve segments in each direction must be specified with `patchprecision`. The precision is the minimum number of line segments approximating each curve segment and can be different for each direction. The actual number of line segments is a multiple of the number of curve segments being drawn in the opposing direction. This guarantees the *u* and *v* curve segments that form the wireframe actually intersect.

5. The surface patch is actually drawn with `patch`. The arguments to `patch` contain the 16 control points that govern the shape of the patch. *geomx* is a 4x4 matrix containing the *x* coordinates of the 16 control points; *geomy* contains the *y* coordinates; *geomz* contains the *z* coordinates. The curve segments in the patch are drawn using the current linestyle, linewidth, color, and writemask.

`rpatch` is the same as `patch`, except it draws a rational surface patch.

patchbasis

`patchbasis` sets the current basis matrices (defined by `defbasis`) for the *u* and *v* parametric directions of a surface patch. `patch` uses the current *u* and *v* when it executes.

```
patchbasis(uid, vid)
long uid, vid;

subroutine patchb(uid, vid)
integer*4 uid, vid

procedure patchbasis(uid, vid: longint);
```

patchcurves

`patchcurves` sets the number of curves used to represent a patch. It sets the current number of *u* and *v* curves that represent a patch as a wire frame.

```
patchcurves(ucurves, vcurves)
long ucurves, vcurves;

subroutine patchc(ucurves, vcurves)
integer*4 ucurves, vcurves

procedure patchcurves(ucurves, vcurves: longint);
```

patchprecision

`patchprecision` sets the precision at which curves defining a wire frame patch are drawn. The *u* and *v* directions for a patch specify the precisions independently. Patch precisions are similar to curve precisions—they specify the minimum number of line segments used to draw a patch.

```
patchprecision(usegments, vsegments)
long usegments, vsegments;

subroutine patchp(usegments, vsegments)
integer*4 usegments, vsegments

procedure patchprecision(usegments, vsegments: longint);
```

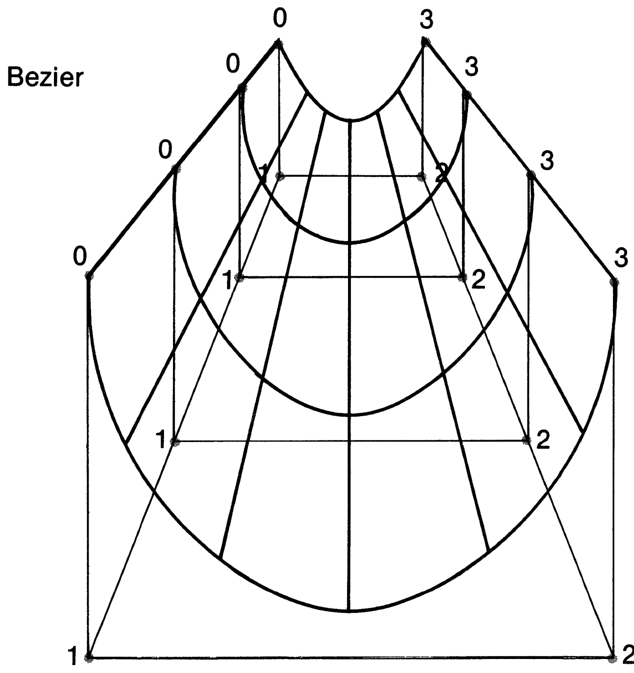
patch, rpatch

`patch` and `rpatch` draw a surface patch using the current `patchbasis`, `patchprecision`, and `patchcurves`. `rpatch` draws a rational surface patch. The control points *geomx*, *geomy*, and *geomz* determine the shape of the patch. *geomw* specifies the rational component of the patch to `rpatch`.

```
patch(geomx, geomy, geomz)
Matrix geomx, geomy, geomz;

rpatch(geomx, geomy, geomz, geomw)
Matrix geomx, geomy, geomz, geomw;

subroutine patch(geomx, geomy, geomz)
real geomx(4,4), geomy(4,4), geomz(4,4)

subroutine rpatch(geomx, geomy, geomz, geomw)
real geomx(4,4), geomy(4,4), geomz(4,4), geomw(4,4)

procedure patch(var geomx, geomy, geomz: Matrix);

procedure rpatch(var geomx, geomy, geomz, geomw: Matrix);
```

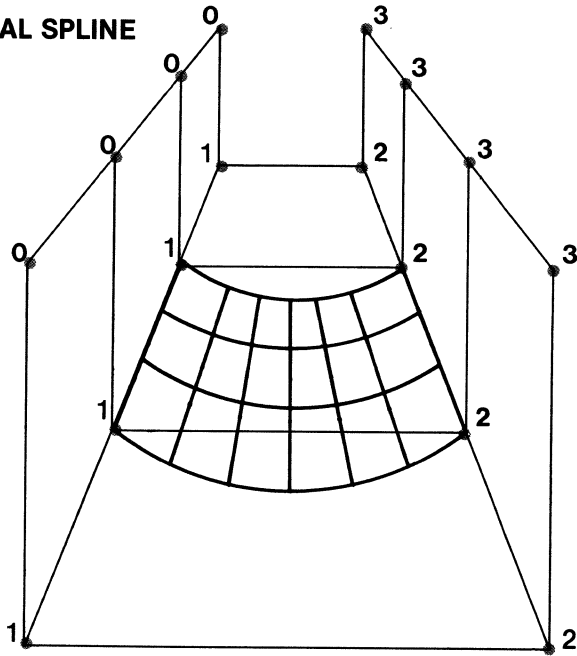
Figure 11-3 shows the same number of curve segments and the same precisions but different basis matrices. All three use the same set of 16 control points.

The following program draws three surface patches similar to those shown in Figure 11-3.



Each patch is drawn using the same set of 16 control points, the same number of curve segments, the same precisions, but different basis matrices.

Figure 11-3. Surface Patches



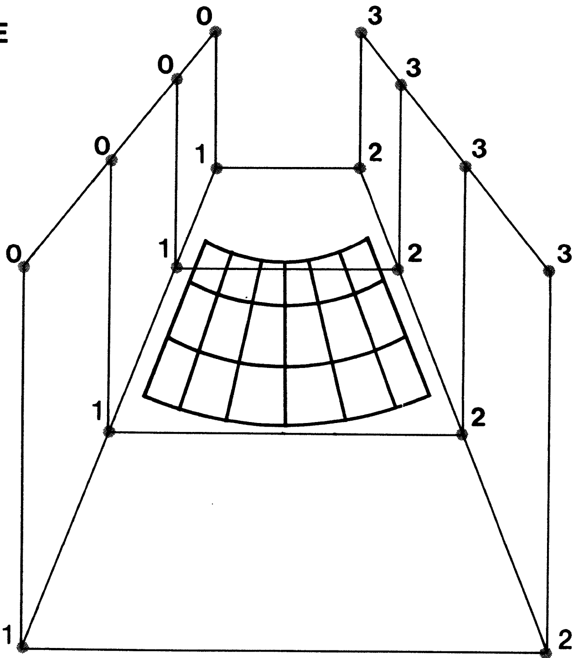


Figure 11-3. Surface Patches (continued)

■ C Program: SURFACE PATCHES

```
#include "gl.h"

Matrix beziermatrix = {
    { -1, 3, -3, 1 },
    { 3, -6, 3, 0 },
    { -3, 3, 0, 0 },
    { 1, 0, 0, 0 }
    };

Matrix cardinalmatrix = {
    { -0.5, 1.5, -1.5, 0.5 },
    { 1.0, -2.5, 2.0, -0.5 },
    { -0.5, 0.0, 0.5, 0.0 },
    { 0.0, 1.0, 0.0, 0.0 }
    };

Matrix bsplinematrix = {
    { -1.0/6.0, 3.0/6.0, -3.0/6.0, 1.0/6.0 },
    { 3.0/6.0, -6.0/6.0, 3.0/6.0, 0.0 },
    { -3.0/6.0, 0.0, 3.0/6.0, 0.0 },
    { 1.0/6.0, 4.0/6.0, 1.0/6.0, 0.0 }
    };

#define BEZIER 1
#define CARDINAL 2
#define BSPLINE 3

Coord geomx[4][4] = {
    { 0.0, 100.0, 200.0, 300.0},
    { 0.0, 100.0, 200.0, 300.0},
    { 700.0, 600.0, 500.0, 400.0},
    { 700.0, 600.0, 500.0, 400.0}
    };

Coord geomy[4][4] = {
    { 400.0, 500.0, 600.0, 700.0},
    { 0.0, 100.0, 200.0, 300.0},
    { 0.0, 100.0, 200.0, 300.0},
    { 400.0, 500.0, 600.0, 700.0}
    };
```

```
Coord geomz[4][4] = {
    { 100.0, 200.0, 300.0, 400.0 },
    { 100.0, 200.0, 300.0, 400.0 },
    { 100.0, 200.0, 300.0, 400.0 },
    { 100.0, 200.0, 300.0, 400.0 }
    };

main ()
{

ginit();
color(BLACK);
clear();

ortho(0.0, (float)XMAXSCREEN, 0.0, (float)YMAXSCREEN,
       (float)XMAXSCREEN, -(float)XMAXSCREEN);

defbasis(BEZIER,beziermatrix); /* define a basis matrix
            called BEZIER */
defbasis(CARDINAL,cardinalmatrix); /* define a basis matrix
            called CARDINAL */
defbasis(BSPLINE,bsplinematrix); /* define a basis matrix
            called BSPLINE */

patchbasis(BEZIER,BEZIER); /* a Bezier basis will be used for both
                directions in the first patch */
    patchcurves(4,7); /* seven curve segments will be drawn in the
                u direction and four in the v direction */
    patchprecision(20,20); /* the curve segments in u direction
                will consist of 20 line segments (the
                lowest multiple of vcurves greater than
                usegments) and the curve segments in
                the v direction will consist of 21
                line segments (the lowest multiple of
                ucurves greater than vsegments) */
color(RED);
    patch(geomx,geomy,geomz); /* the patch is drawn based on the
                sixteen specified control points */

    patchbasis(CARDINAL,CARDINAL); /* the bases for both
                    directions are reset */
```

```
    color(GREEN);
    patch(geomx,geomy,geomz);  /* another patch is drawn using
              the same control points but a
              different basis */

    patchbasis(BSPLINE,BSPLINE);  /* the bases for both
              directions are reset again */
    color(BLUE);
    patch(geomx,geomy,geomz);  /* a third patch is drawn */

sleep(10);
gexit();
}
```

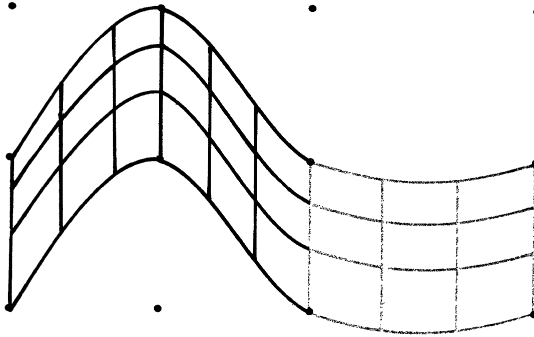
■ FORTRAN Program: SURFACE PATCHES

```
      #INCLUDE 'fgl.h'
C
      REAL BEZMAT(4,4), CARMAT(4,4), BSPMAT(4,4)
      REAL GEOMX(4,4), GEOMY(4,4), GEOMZ(4,4)
      INTEGER*2 BEZI, CARD, BSPL
      REAL ASIXTH, MSIXTH, TTHRDS, XMAX, YMAX
      INTEGER I, NOP
      PARAMETER ( BEZIER = 1 )
      PARAMETER ( CARDIN = 2 )
      PARAMETER ( BSPLIN = 3 )
      PARAMETER ( ASIXTH = 1.0/6.0 )
      PARAMETER ( MSIXTH = -1.0/6.0 )
      PARAMETER ( TTHRDS = 2.0/3.0 )
C
      DATA BEZMAT /-1.0,  3.0, -3.0,  1.0,
     2             3.0, -6.0,  3.0,  0.0,
     3            -3.0,  3.0,  0.0,  0.0,
     4             1.0,  0.0,  0.0,  0.0/
      DATA CARMAT /-0.5,  1.5, -1.5,  0.5,
     2             1.0, -2.5,  2.0, -0.5,
     3            -0.5,  0.0,  0.5,  0.0,
     4             0.0,  1.0,  0.0,  0.0/
      DATA BSPMAT /MSIXTH,    0.5,    -0.5, ASIXTH,
     2                0.5,   -1.0,     0.5,    0.0,
```

```
     3                -0.5,    0.0,    0.5,    0.0,
     4              ASIXTH, TTHRDS, ASIXTH,    0.0/
      DATA GEOMX /  0.0, 100.0, 200.0, 300.0,
     2              0.0, 100.0, 200.0, 300.0,
     3            700.0, 600.0, 500.0, 400.0,
     4            700.0, 600.0, 500.0, 400.0/
      DATA GEOMY /400.0, 500.0, 600.0, 700.0,
     2              0.0, 100.0, 200.0, 300.0,
     3              0.0, 100.0, 200.0, 300.0,
     4            400.0, 500.0, 600.0, 700.0/
      DATA GEOMZ /100.0, 200.0, 300.0, 400.0,
     2            100.0, 200.0, 300.0, 400.0,
     3            100.0, 200.0, 300.0, 400.0,
     4            100.0, 200.0, 300.0, 400.0/
C
      CALL GINIT
      CALL COLOR(BLACK)
      CALL CLEAR()
      XMAX = XMAXSC
      YMAX = YMAXSC
      CALL ORTHO(0.0,XMAX,0.0,YMAX,XMAX,-XMAX)
C       define a basis matrix called BEZIER
      CALL DEFBAS(BEZIER,BEZMAT)
C       define a basis matrix called CARDIN
      CALL DEFBAS(CARDIN,CARMAT)
C       define a basis matrix called BSPLIN
      CALL DEFBAS(BSPLIN,BSPMAT)
C
C       a Bezier basis will be used for both directions in the
         first patch
      CALL PATCHB(BEZIER,BEZIER)
C       7 curve segments will be drawn in the u direction
C       and 4 in the v direction
      CALL PATCHC(4,7)
C       the curve segments in the u direction will consist of
        20 line
C       segments (the lowest multiple of vcurves greater than
         usesegments)
C       & the curve segments in the v direction will consist
         of 21 line
C       segments (the lowest multiple of ucurves greater than
```

```
           vsegments)
      CALL PATCHP(20,20)
      CALL COLOR(RED)
C       the patch is drawn based on the 16 specified control
           points
      CALL PATCH(GEOMX,GEOMY,GEOMZ)
C
C       reset the bases for both directions
      CALL PATCHB(CARDIN,CARDIN)
      CALL COLOR(GREEN)
C       draw another patch using the same control points
C       but a different basis
      CALL PATCH(GEOMX,GEOMY,GEOMZ)
C
C       reset the bases for both directions again
      CALL PATCHB(BSPLIN,BSPLIN)
      CALL COLOR(BLUE)
C       draw a third patch
      CALL PATCH(GEOMX,GEOMY,GEOMZ)
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C      Installation note:
C       This is a delay loop, which may require adjustment.
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      CALL GFLUSH()
      DO 1000 I=1,1000000
      NOP = I * 2
1000  CONTINUE
      CALL GEXIT()
      STOP
      END
```

You can join patches together to create a more complex surface by using the Cardinal spline or B-spline bases and overlapping sets of control points. The surface in Figure 11-4 consists of three joined patches and was drawn using a B-spline basis.



The above surface consists of three joined patches. The patches are drawn with overlapping sets of control points, using a Cardinal spline basis matrix.

Figure 11-4. Joined Patches

12. Hidden Surfaces

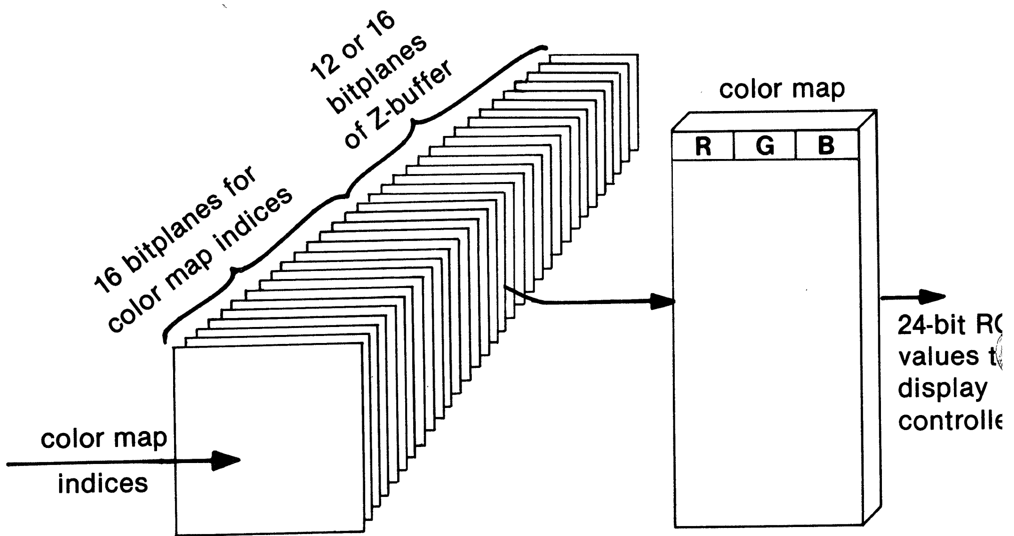
Chapter 4, Coordinate Transformations, tells how to create 3-D models in world space and how to project images of those models onto the screen. This chapter discusses how to make those images appear more realistic by removing the hidden lines and surfaces from the images using z-buffering and backfacing polygon removal.

12.1 Z-Buffer Mode

When the IRIS is in z-buffer mode, the *z* coordinate for each pixel on the screen is stored in the bitplanes. (See Chapter 6, Display and Color Modes, for a discussion of bitplanes and display modes.) When a pixel is drawn, its new z value is compared to the existing z value. If the new z value is less than or equal to the existing z value (i.e., closer to the viewer), the new color value for that pixel is written into the bitplanes, along with the new z value. Otherwise, the color and z values are unchanged. As a result, only the parts of the image visible to the viewer are displayed on the screen.

Z-buffering only works in color map, single buffer display mode. The z-buffer bits occupy the 16 high-order bitplanes of a 32-bitplane system and the 12 high-order bitplanes of a 28-bitplane system. (See Figure 12-1.) Z-buffering is not effective in a system with less than 28 bitplanes.

`setdepth`, `zclear`, and `zbuffer` set up z-buffering. `getdepth` and `getzbuffer` make inquiries about z-buffering.



In z-buffer mode, the low-order 12 bitplanes store the color map indices for each pixel, and the high-order 16 bitplanes store the corresponding z values for each pixel. In color map mode, color indices store only 12 of the 16 available bitplanes.

Figure 12-1. Z-Buffer Bitplanes

setdepth

`setdepth` sets the scaling of the near and far clipping planes. `setdepth` specifies the range of z values that are stored in bitplanes. If there are 16 bits of z-buffer, set *near* to 0xC000 and set *far* to 0x3FFF. If there are 12 bits of z-buffer, set *near* to 0 and set *far* to 0xFFF.

```
setdepth(near,far)
Screencoord near,far;

subroutine setdep(near,far)
integer*4 near,far

procedure setdepth(near,far: longint);
```

getdepth

`getdepth` returns the *near* and *far* values set by `setdepth`.

```
getdepth(near,far)
Screencoord *near, *far;

subroutine getdep(near,far)
integer*4 near,far

procedure getdepth(var near,far: Screencoord);
```

zclear

`zclear` initializes the z-buffer to the largest positive integer (0x7FFF). Any point with a z coordinate less than or equal to the current z value is drawn to the screen.

```
zclear()

subroutine zclear

procedure zclear;
```

zbuffer

`zbuffer(TRUE)` turns on z-buffer mode; `zbuffer(FALSE)` turns it off. FALSE(0) is the default value. Z-buffering does not work for lines that have a width greater than 1.

```
zbuffer(bool)
Boolean bool;

subroutine zbuffe(bool)
logical bool

procedure zbuffer(bool: longint);
```

getzbuffer

`getzbuffer` indicates whether z-buffering is on or off. FALSE(0) is the default value and means that z-buffering is off. TRUE(1) means that z-buffering is on.

```
long getzbuffer()

logical function getzbu()

function getzbuffer: longint;
```

The following program draws two intersecting polygons in z-buffer mode in a system with 28 bitplanes. Only the 'visible' part of each polygon appears on the screen:

■ C Program: Z-BUFFER

```
#include "gl.h"

main ()
{
ginit();
color(BLACK);
clear();
ortho(0.0, (float)XMAXSCREEN, 0.0, (float)YMAXSCREEN,
       0.0, -(float)XMAXSCREEN);
setdepth(0x000,0xFFF);  /* the minimum and maximum z values
     are set */

zbuffer(TRUE);  /* the IRIS enters z-buffering mode */
```

```
zclear();  /* the z-buffer is cleared to the maximum z value */

color(YELLOW);     /* draw the first polygon in yellow */
pmv(0.0, 0.0, 100.0);
pdr(100.0, 0.0, 100.0);
pdr(100.0, 100.0, 100.0);
pdr(0.0, 100.0, 100.0);
pclos();

color(RED);     /* draw the second polygon in red */
pmv(0.0, 0.0, 50.0);
pdr(100.0, 0.0, 50.0);
pdr(100.0, 100.0, 200.0);
pdr(0.0, 100.0, 200.0);
pclos();

zbuffer(FALSE);  /* the IRIS exits z-buffering mode */
gexit();
}
```

■ FORTRAN Program: Z-BUFFER

```
     #INCLUDE'/usr/include/fgl.h
     #include /usr/include/fdevice.h

      call ginit
      call color(BLACK)
      call clear
      xmax=xmaxsc
      ymax=ymaxsc

      call ortho(0.0,xmax,0.0,ymax,0.0,-xmax)
      call setdep($000,$fff)
      call zbuffe(.TRUE.)
      call zclear

c*** draw the first polygon in yellow
c*** not drawing
```

```
      call color(YELLOW)
      call pmv(0.0,0.0,100.0)
      call pdr(100.0,0.0,100.0)
      call pdr(100.0,100.0,100.0)
      call pdr(0.0,100.0,100.0)
      call pclos

c*** draw the second polygon in red

      call color(RED)
      call pmv(0.0,0.0,50.0)
      call pdr(100.0,0.0,50.0)
      call pdr(100.0,100.0,200.0)
      call pdr(0.0,100.0,200.0)
      call pclos
      call zbuffe(.FALSE.)

 999  continue
      if(.not. getbut(RIGHTM))go to 999
      call gexit
      stop
      end
```

12.2 Backfacing Polygon Removal

A backfacing polygon is defined as a polygon whose vertices appear in clockwise order in screen space. When backfacing polygon removal is turned on, only polygons whose vertices appear in counterclockwise order are displayed, i.e., polygons that point toward you. Therefore, the vertices of all polygons should be specified in counterclockwise order.

backface

`backface` initiates or terminates backfacing polygon removal. The backface utility is used to improve the performance of programs that represent solid shapes as collections of polygons. The vertices of the polygons on the 'far' side of the solid are in clockwise order and are not drawn.

Backfacing polygon removal is not a reliable technique for hidden surface removal because some frontfacing polygons can also be obscured. When a polygon shrinks to the point where its vertices are coincident, its orientation is indeterminate and it is displayed. Thus, backfacing polygon removal should be used in conjunction with some other hidden surface removal technique, such as z-buffering. (See Section 12.1.)

Backface removal is useful for simple convex objects. For more general images, hidden surface removal must be accomplished using another technique, perhaps in conjunction with backface removal.

In backface mode, certain matrices, such as scale(-1.0, 1.0, 1.0) reverse the sense of rectangles and polygons.

```
backface(b)
Boolean b;

subroutine backfa(b)
logical b

procedure backface(b: longint);
```

getbackface

`getbackface` returns the state of backfacing filled polygon removal. If backface removal is on, the system draws only those polygons that face the viewer. If backfacing polygon removal is enabled, 1 is returned; otherwise, 0 is returned.

```
long getbackface()

integer*4 getbac()

function getbackface(): longint;
```

The following example draws a cube in which only the visible surfaces are drawn. The cube can be rotated by moving the mouse.

■ C Program: BACKFACE REMOVAL

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

#define CUBE_SIZE 200

main()
{

    int foo;

    ginit();
    doublebuffer();
    gconfig();

    viewport(0, YMAXSCREEN, 0, YMAXSCREEN);

    ortho(-YMAXSCREEN.0, YMAXSCREEN.0,
        -YMAXSCREEN.0, YMAXSCREEN.0,
        -YMAXSCREEN.0, YMAXSCREEN.0);

    qdevice(KEYBD);
    makeobj(1);
    rectfi(-CUBE_SIZE,-CUBE_SIZE,CUBE_SIZE,CUBE_SIZE);
    closeobj();

    /* define a cube */
    makeobj(2);
    /* front face */
    pushmatrix();
    translate(0.0,0.0,CUBE_SIZE.0);
    color(RED);
    callobj(1);
    popmatrix();

    /* right face */
    pushmatrix();
    translate(CUBE_SIZE.0, 0.0, 0.0);
    rotate(900, 'y');
    color(GREEN);
    callobj(1);
```

```
popmatrix();

/* backface */
pushmatrix();
translate(0.0, 0.0, -CUBE_SIZE.0);
rotate(1800, 'y');
color(BLUE);
callobj(1);
popmatrix();

/* left face */
pushmatrix();
translate(-CUBE_SIZE.0, 0.0, 0.0);
rotate(-900, 'y');
color(CYAN);
callobj(1);
popmatrix();

/* top face */
pushmatrix();
translate(0.0, CUBE_SIZE.0, 0.0);
rotate(-900, 'x');
color(MAGENTA);
callobj(1);
popmatrix();

/* bottom face */
pushmatrix();
translate(0.0, -CUBE_SIZE.0, 0.0);
rotate(900, 'x');
color(YELLOW);
callobj(1);
popmatrix();

closeobj();

/* turn on back facing polygon removal */
backface(1);

while ((foo=qtest()) !=KEYBD) {
  if(foo) qreset();
```

```
    pushmatrix();
    rotate(2*getvaluator(MOUSEX), 'x');
    rotate(2*getvaluator(MOUSEY), 'y');
    color(BLACK);
    clear();
    callobj(2);
    popmatrix();
    swapbuffers();
  }

  backface(0);

  gexit();
}
```

■ FORTRAN Program: BACKFACE REMOVAL

```
#INCLUDE /usr/include/fgl.h
#INCLUDE /usr/include/fdevice.h

      integer cubesz
      real    ymax
      integer foo

      data cubesz/200/

      call ginit
      call double
      call gconfi
      call viewpo(0,ymaxsc,0,ymaxsc)
      ymax=ymaxsc*1.0
      qbsize=cubesz*1.0

      call ortho(-ymax,ymax,-ymax,ymax,-ymax,ymax)
```

```
      call qdevic(KEYBD)
      call makeob(1)
       call rectfi(-cubesz,-cubesz,cubesz,cubesz)
      call closeo

c*  define a cube

      call makeob(2)
c
c***   front face
c
       call pushma
       call transl(0.0,0.0,qbsize)
       call color(RED)
       call callob(1)
       call popmat

c***   right face

       call pushma
       call transl(qbsize,0.0,0.0)
       call rotate(900,'y')
       call color(GREEN)
       call callob(1)
       call popmat

c***   back face

       call pushma
       call transl(0.0,0.0,-qbsize)
       call rotate(1800,'y')
       call color(BLUE)
       call callob(1)
       call popmat
```

```
c***  left face

       call pushma
       call transl(-qbsize,0.0,0.0)
       call rotate(-900,'y')
       call color(CYAN)
       call callob(1)
       call popmat

c***  top face

       call pushma
       call transl(0.0,qbsize,0.0)
       call rotate(-900,'x')
       call color(WHITE)
       call callob(1)
       call popmat

c***  bottom face

       call pushma
       call transl(0.0,-qbsize,0.0)
       call rotate(900,'x')
       call color(YELLOW)
       call callob(1)
       call popmat
      call closeo

c*** turn on back facing polygon removal

      call backfa(1)
 15   continue
      foo = qtest()
      if (foo .eq. KEYBD) then
      go to  999
      end if
      if (foo .eq. 1) then
        call qreset
      end if
      call pushma
      call rotate(2*getval(MOUSEX),'x')
```

```
      call rotate(2*getval(MOUSEY),'y')
      call color(BLACK)
      call clear
      call callob(2)
      call popmat
      call swapbu
      go to 15
 999  continue
      call backfa(0)
      call gexit
      stop
      end
```

13. Shading and Depth-Cueing

The previous chapter discussed how to make an image appear more realistic by removing the hidden lines and surfaces from the image. This chapter discusses two more techniques that can make an image appear more realistic:

- shading the polygons in an image
- depth-cueing the polygons, lines, and points in an image (i.e., varying their intensities based on their z values)

13.1 Shading

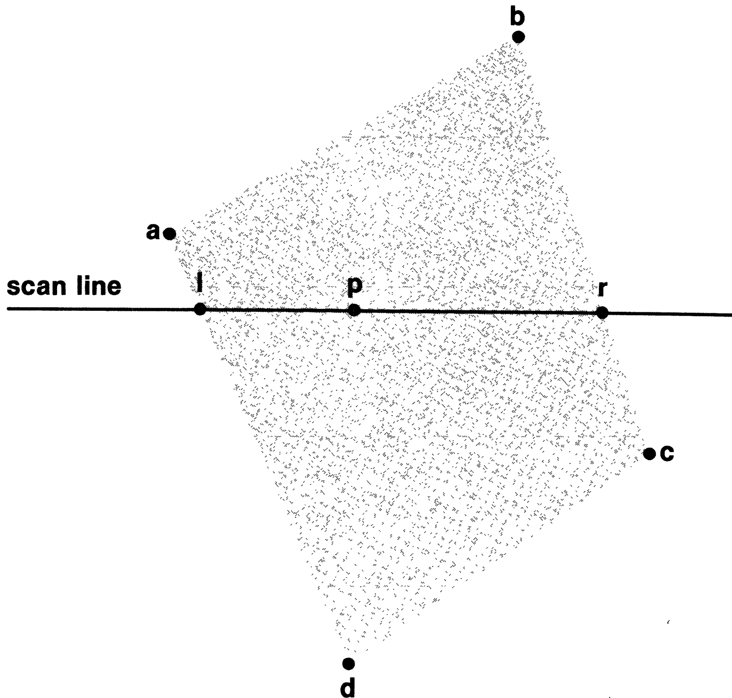
The IRIS shading facility provides *Gouraud* shading of polygons. This shading technique uses linear interpolation to determine the intensities of each pixel in a filled polygon. You control the shading of a polygon by specifying the intensities of each of the polygon's vertices. The intensity of a vertex is a color map index. The range of desired intensities must be reflected by the values contained in the color map. For example, if the vertex at one side of a polygon were assigned a color map index that contained a dark blue color and the vertex at the other side were assigned an index with a light blue color, then all of the indices between the two should be loaded with intermediate shades of blue.

splf

`splf` draws a shaded polygon. The first two arguments of `splf` are the same as the first two arguments of `polf`. (See Chapter 3, Drawing Routines, Section 3.6.) *n* specifies the number of vertices in the polygon. *parray* is the array of vertices. The third argument, *iarray*, is an array of *n* color map indices (one for each vertex).

```
splf(n, parray, iarray)
long n;
Coord parray[][3];
Colorindex iarray[];

subroutine splf(n, parray, iarray)
integer*4 n
real parray(3,n)
integer*2 iarray(n)

procedure splf(n: longint; var parray: Coord;
     var iarray: Colorindex);
```

Figure 13-1 illustrates how Gouraud shading works. Each vertex of a polygon is assigned an intensity. The intensities for the points along the edges of the polygon are determined by interpolating linearly between the intensities at the vertices. The intensities for all of the interior points of the polygon are determined by interpolating linearly between the pairs of edge-points that lie along each scan line. The entries in the color map and the values assigned to the vertices of each polygon must be coordinated to create the desired shading effect.



$$c_l = c_d + \left(\frac{l_y - d_y}{a_y - d_y} \right) \left(c_a - c_d \right)$$

$$c_r = c_c + \left(\frac{r_y - c_y}{b_y - c_y} \right) \left(c_b - c_c \right)$$

$$c_p = c_l + \left(\frac{r_x - p_x}{r_x - l_x} \right) \left(c_r - c_l \right)$$

The intensities (color map indices) for points l and r are determined by linear interpolation between the intensities at the endpoints of their edges. The intensity at point p is determined by linear interpolation between between the intensities at point l and point r.

Figure 13-1. Gouraud Shading

setshade

`setshade` assigns an intensity to the vertex specified in the routine that immediately follows. Use `setshade` in conjunction with `pmv` and `pdr`. If `pclos` is called, a filled and unshaded polygon is drawn in the current color and `setshade` is ignored.

Note: Do not use `setshade` under the window manager.

```
setshade(shade)
Colorindex shade;

subroutine setsha(shade)
integer*4 shade

procedure setshade(shade: longint);
```

spclos

`spclos` closes and shades the polygon.

```
spclos()

subroutine spclos

procedure spclos;
```

getshade

`getshade` returns the current color. It is an index into the color map and is useful only in color map mode.

```
long getshade()

integer*4 function getsha()

function getshade: longint;
```

The following program draws a shaded polygon using `pmv`, `pdr`, and `setshade`. Note that shading works only in single buffer and double buffer display modes, and not in RGB mode.

■ C Program: SHADED POLYGON

```
#include "gl.h"

main ()
```

```
{

int i;

ginit();
color(BLACK);
clear();

/* create a magenta color ramp */
for (i = 0; i < 128; i++)
    mapcolor(i, 2*i, 0, 2*i);
setshade(0);        /* set the intensity for the first vertex */
pmv(100.0,100.0,0.0);      /* specify the first vertex */
setshade(127);      /* set the intensity for the second vertex */
pdr(200.0,200.0,0.0);      /* specify the second vertex */
setshade(64);           /* set the intensity for the third vertex */
pdr(200.0,100.0,0.0);      /* specify the third vertex */
spclos();       /* close the shaded polygon */

sleep(5);
greset();
gexit();

}
```

■ FORTRAN Program: SHADED POLYGON

```
     #INCLUDE /usr/include/fgl.h
     #INCLUDE /usr/include/fdevice.h

      integer i

      call ginit
      call color(BLACK)
      call clear
      do 10 i = 0 , 127
        call mapcol(i,2*i,0,2*i)
 10   continue

      call setsha(0)
```

```
      call pmv(100.0,100.0,0.0)
      call setsha(127)
      call pdr(200.0,200.0,0.0)
      call setsha(64)
      call pdr(200.0,100.0,0.0)
      call spclos
 99   continue
      if (.not. getbut(RIGHTM) ) go to 99
      call greset
      call color(BLACK)
      call clear
      call gexit
      stop
      end
```

13.2 Depth-Cueing

Depth-cueing is typically used with wireframe models to help the eye discern 3-D characteristics by making lines brighter that are closer to the eye, and lines that are farther away less bright. In depth-cue mode, the intensities of all the polygons, lines, and points drawn to the screen vary according to their z values.

depthcue

`depthcue` turns depth-cue mode on and off. If *mode* is TRUE(1), all lines, points, characters, and polygons that the system draws are depth-cued. When *mode* is FALSE(0), depth-cue mode is off. For depth cueing to work, the color map locations that `shaderange` specifies must be loaced with a series of colors that gradually increase or decrease intensity.

```
depthcue(mode)
short mode;

subroutine depthc(mode)
logical mode

procedure depthcue(mode: longint);
```

getdcm

`getdcm` indicates whether depth-cue mode is on or off. TRUE(1) means depth-cue mode is on and FALSE(0) means depth-cue mode is off.

```
long getdcm()
```

```
logical function getdcm()
```

```
function getdcm: longint;
```

shaderange

`shaderange` specifies the low-intensity color map index (*lowindex*) and the high-intensity color map index (*highindex*). These values are mapped to the low and high z values specified by *z1* and *z2*. The high index must be greater than the low index and the difference between the high index and the low index must be less than the difference between *z1* and *z2*. The values of *z1* and *z2* should correspond to or lie within the range of z values specified by `setdepth`. `setdepth` is the entire transformation range. `shaderange` is the range of values where all shading occurs.

`shaderange` doesn't work in RGB mode.

```
shaderange(lowindex, highindex, z1, z2)
Colorindex lowindex, highindex;
Screencoord z1, z2;
```

```
subroutine shader(lowindex, highindex, z1, z2)
integer*4 lowindex, highindex, z1, z2
```

```
procedure shaderange(lowindex,highindex: longint;
     z1, z2: longint);
```

The entries for the color map between the low index and the high index should reflect the appropriate sequence of intensities for the color being drawn. When a depth-cued point is drawn, its z value is used to determine its intensity. When a depth-cued line is drawn, the intensities of its points are linearly interpolated from the intensities of its endpoints, which are determined from their z values. You can achieve higher resolution if the near and far clipping planes bound the object as closely as possible. The following equation yields the color map index for a point with *z* coordinate *z*:

$$
color_z = \begin{cases} highindex, & \text{if } z \le z1; \\ (\frac{highindex - lowindex}{z2 - z1})(z - z1) + lowindex, & \text{if } z1 \le z \le z2; \\ lowindex, & \text{if } z2 \le z. \end{cases}
$$

Note that this equation yields a nonlinear mapping when z is less than *z1* or greater than *z2* and the color is limited to [*highindex, lowindex*]. Since depth-cued lines are linearly interpolated between endpoints, an endpoint outside the range of *z1* and *z2* can result in an undesirable image.

The following program draws a cube filled with points, which rotates as the mouse is moved. Since the image is drawn in depth-cue mode, the edges of the cube and the points inside the cube that are closer to the viewer are brighter than the edges and points farther away from the viewer.

■ C Program: DEPTH-CUED CUBE

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

float hrand();

main ()
{
    int val;
    int i;

    ginit();
    doublebuffer();
    gconfig();

    ortho(-350.0, 350.0,
      -350.0, 350.0,
      -350.0, 350.0);
    viewport(0, YMAXSCREEN, 0, YMAXSCREEN);
    qdevice(KEYBD);

    makeobj(1);

    /* a bunch of random points */
```

```
for (i = 0; i < 100; i++)
    pnt(hrand(-200.0,200.0), hrand(-200.0,200.0),
            hrand(-200.0,200.0));

/* and a cube */
movei(-200, -200, -200);
drawi(200, -200, -200);
drawi(200, 200, -200);
drawi(-200, 200, -200);
drawi(-200, -200, -200);
drawi(-200, -200, 200);
drawi(-200, 200, 200);
drawi(-200, 200, -200);
movei(-200, 200, 200);
drawi(200, 200, 200);
drawi(200, -200, 200);
drawi(-200, -200, 200);
movei(200, 200, 200);
drawi(200, 200, -200);
movei(200, -200, -200);
drawi(200, -200, 200);
closeobj();

/* load the color map with a cyan ramp */
for (i = 0; i < 128; i++)
mapcolor(128+i, 0, 2*i, 2*i);
/* set the range of z values that will be stored
in the bitplanes */
setdepth(0xC000, 0x3fff);
/* set up the mapping of z values to color map indices:
z value 0 is mapped to index 128 and z value 0x7fff is
mapped to index 255 */
shaderange(128,255,0,0x7fff);

/* turn on depthcue mode:  the color index of each pixel in
points and lines is determined from the z value of the pixel */
depthcue(1);
/* until a key is pressed, rotate the cube according to the
movement of the mouse */
while ((val=qtest()) != KEYBD) {
pushmatrix();
```

```
    rotate(3*getvaluator(MOUSEY), 'x');
    rotate(3*getvaluator(MOUSEX), 'y');
    color(BLACK);
    clear();
    callobj(1);
    popmatrix();
    swapbuffers();
    }

    gexit();
}


/* this routine returns random numbers in the specified range */
float hrand(low,high)
    float low,high;
{
    float val;
    val = ((float)( (short)rand(0) & 0xffff)) / ((float)0xffff);
    return( (2.0 * val * (high-low)) + low);
}
```

■ FORTRAN Program: DEPTH-CUED CUBE

```
      #INCLUDE /usr/include/fgl.h
      #INCLUDE /usr/include/fdevice.h
c
c*     convert from p13-5 of User's Guide
c
       integer val
       integer i
       real    hrand,ranval

       call ginit
       call double
       call gconfi

       call ortho(-350.0,350.0,-350.0,350.0,-350.0,-350.0)
       call viewpo(0,ymaxsc,0,ymaxsc)
       call qdevic(KEYBD)
```

```
      call makeob(1)
       do 10 i = 1,100
          ranval = hrand(-200.0,200.0)
          call pnt(ranval,ranval,ranval)
 10        continue
       call movei(-200,-200,-200)
       call drawi(200,-200,-200)
       call drawi(200,200,-200)
       call drawi(-200,200,-200)
       call drawi(-200,-200,-200)
       call drawi(-200,-200,200)
       call drawi(-200,200,200)
       call drawi(-200,200,-200)
       call movei(-200,200,200)
       call drawi(200,200,200)
       call drawi(200,-200,200)
       call drawi(-200,-200,200)
       call movei(200,200,200)
       call drawi(200,200,-200)
       call movei(200,-200,-200)
       call drawi(200,-200,200)
      call closeo

c*** load the color map with a cyan ramp

      do 20 i = 0,127
        call mapcol(128+i , 0 , 2*i ,2*i)
 20   continue

c*** set the range of z values that will be stored in the bitplanes

      call setdep($C000, $7fff)

c*** set up the mapping of z values to color map indices:
c*** z value 0 is mapped to index 128 and z value $7fff is
c*** mapped to index 255

      call shader(128,255,0,$7fff)
```

```
c*** turn on depthcue mode: the color index of each pixel in point:
c*** and lines is determined from the z value of the pixel

      call depthc(1)

c*** until a key is pressed, rotate the cube according to the
c*** movement of the mouse
 25   continue
      val = qtest()
      if ( val .eq. KEYBD)go to 99
      call pushma
      call rots(3*getval(MOUSEY),'x')
      call rotate(3*getval(MOUSEX),'y')
      call color(0)
      call clear
      call callob(1)
      call popmat
      call swapbu
      go to 25
 99   continue
      call gexit
      stop
      end
      function hrand(low,high)
      real low,high
      real hrand

      hrand= (2*ran(0)*(high-low))+low
      end
```

14. Textports

The textport is an area of the screen that displays text from programs. `printf()` prints characters to the textport. `charstr` draws character strings to the viewport. A default textport (80 columns wide and 40 lines high) appears on the screen at (148, 875, 80, 687) when the IRIS is booted.

Note: Textports are not defined under the window manager.

tpon

`tpon` turns on the textport. The textport turns on automatically when you exit a program in which `tpoff` has been called.

```
tpon()

subroutine tpon

procedure tpon;
```

tpoff

`tpoff` turns off the textport. When the textport is off, characters are not written to the textport and the textport does not appear on the screen.

```
tpoff()

subroutine tpoff

procedure tpoff;
```

textport

`textport` changes the size and the location of the textport. It allocates an area of the screen for the textport by specifying the left, right, bottom, and top edges of the textport.

`textport(0,0,0,0)` turns off the textport; it will not appear on the screen until the textport size is reset or `textinit` (see below) is called.

Note: `texport` does not work in RGB mode.

```
textport(left, right, bottom, top)
Screencoord left, right, bottom, top;

subroutine textpo(left, right, bottom, top)
integer*4 left, right, bottom, top

procedure textport(left, right, bottom, top: longint);
```

gettp

`gettp` returns the current textport size and location (*left, right, bottom,* and *top* edges).

```
gettp(left, right, bottom, top)
Screencoord *left, *right, *bottom, *top;

subroutine gettp(left, right, bottom, top)
integer*2 left, right, bottom, top

procedure gettp(var left, right, bottom, top: Screencoord);
```

textcolor

`textcolor` sets the color of the text drawn in the textport. *tcolor* determines the color of text written in the viewport with `charstr`.

`textcolor` is undefined under the window manager.

```
textcolor(tcolor)
Colorindex tcolor;

subroutine textco(tcolor)
integer*4 tcolor

procedure textcolor(tcolor: longint);
```

textwritemask

`textwritemask` sets the writemask for text drawn in the textport. It does not affect text drawn with `charstr`.

```
textwritemask(tmask)
Colorindex tmask;

subroutine textwr(tmask)
integer*4 tmask

procedure textwritemask(tmask: longint);
```

pagecolor

`pagecolor` sets the color for the background of the textport.

```
pagecolor(c)
short c;

subroutine pageco(c)
integer*4 c

procedure pagecolor(c: longint);
```

pagewritemask

`pagewritemask` sets the writemask for the background of the textport.

```
pagewritemask(pmask)
short pmask;

subroutine pagewr(pmask)
integer*4 pmask

procedure pagewritemask(pmask: longint);
```

textinit

`textinit` resets the textport to its default values.

```
textinit()

subroutine textin

procedure textinit;
```

USING MEX, THE IRIS WINDOW MANAGER

1. Getting Started with *mex* W-1
- 1.1 What Is a Window Manager? W-1
- 1.2 Window Manager Terminology W-2
- 1.3 What Does *mex* Do? W-3
- 1.4 Creating Windows W-3
- 1.5 Interacting with the Window Manager W-4
  - 1.5.1 Attaching to Windows W-5
  - 1.5.2 Selecting a Window W-5
  - 1.5.3 Moving a Window W-5
  - 1.5.4 Reshaping a Window W-6
  - 1.5.5 Pushing and Popping Windows W-6
  - 1.5.6 Removing a Window W-6
- 1.6 Selecting a Title Font W-7
- 1.7 Attaching to the Window Manager W-7
- 1.8 Executing Graphics Programs from *mex* W-8

2. Programming with *mex* W-9
- 2.1 Opening and Closing Windows W-9
- 2.2 Setting Window Constraints W-11
  - 2.2.1 Setting Constraints for Existing Windows W-16
- 2.3 Changing Windows Noninteractively (from within a Program) W-17
- 2.4 Other Window Routines W-19
- 2.5 Programming Hints W-24
  - 2.5.1 Graphics Initialization W-24
  - 2.5.2 Shared Facilities W-24
  - 2.5.3 Raster Fonts W-24
  - 2.5.4 The Event Queue W-25
- 2.6 Sample Program: Single Buffer Mode W-25
- 2.7 Sample Program: Double Buffer Mode W-27

3. Making Pop-up Menus W-31
3.1 Creating a Pop-up Menu W-32
3.2 Calling Up a Pop-up Menu W-34
3.3 Choosing Colors for Pop-up Menus W-36
3.4 Advanced Menu Formats W-38
3.4.1 Getting Back the Default Values for Menu Selections W-38
3.4.2 Changing the Return Values for Menu Selections W-39
3.4.3 Making a Title W-39
3.4.4 Binding a Function to a Whole Menu W-39
3.4.5 Binding a Function to a Menu Entry W-40
3.4.6 Making a Nested (Rollover) Menu W-40
3.5 An Example from *cedit*, a Color Editing Program W-41
3.6 Sample Program W-43

4. Customizing *mex* W-47
4.1 Interpreting Button Events W-48
4.2 Binding Colors to the Title Bar and Borders W-51
4.3 Binding Colors to Pop-up Menus and the Cursor W-53
4.4 Setting Color Map Entries W-53
4.5 A Sample *.mexrc* W-53
4.6 Arranging Your Desktop W-54

5. Controlling Multiple Windows from a Single Process W-57

1. Getting Started with *mex*

mex is the window manager for the IRIS workstation; it is an acronym for *m*ultiple *ex*posure. With *mex*, you can create several independent graphics displays, or *windows*, on the screen of an IRIS workstation.

This chapter describes the default configuration of the window manager. Chapter 4, Customizing *mex*, tells how to change the configuration. If the window manager you are running does not display the same behavior as the default configuration described below, then the configuration on your workstation has been altered.

1.1 What Is a Window Manager?

A window manager is a user interface environment that controls multiple processes. Window managers include the following features:

- separate areas of a workstation screen to do parallel tasks simultaneously. These areas are generally called windows and can be arranged in different ways. For example, tiled windows are always adjacent to one another, and overlapping windows can be stacked, so some of the windows are not in full view.
- mouse control to select items from various pop-up menus, which activate different windows, change their position, or otherwise affect them
- configuration of windows, that is, the user can change the behavior of the input device and the color of the windows

The window manager environment has its own terminology that describes its various concepts and functions. This terminology is defined in the section below.

1.2 Window Manager Terminology

This section describes the terms and concepts associated with the IRIS window manager which you will find throughout this document.

Each *window* provides an individual graphics context, e.g., a virtual graphics device, which is independent of all other windows. There are two types of windows: text windows, in which you edit text and enter operating system commands, and graphics windows, in which compiled graphics programs are displayed.

A *graphics program* can draw graphics into one or more windows; it can also read input devices and create graphic images. An interactive program can make requests to change the position or appearance of a window. You can control *mex* using a procedural interface, i.e., certain routines from the Graphics Library.

The window with the *input focus*, i.e., the *attached window*, is the window that currently receives input events, such as keystrokes, mouse position, and mouse button events, which go to the process.

The *cursor* is an image on the screen that follows the position of the mouse or other input device. A graphics program can change the current shape of the cursor, which is controlled by the process with the current input focus. The default cursor is an arrow pointing towards the upper-left. The window manager cursor (which you use to manipulate windows) represents three overlapping windows with an arrow.

Graphics programs and the window manager use *pop-up menus*, which allow you to select (activate) an item on the menu. For example, the window menu provides items such as 'attach' and 'move', which allow you to direct the input focus to an appropriate window and to physically move the window around the screen, respectively. You can create pop-up menus in your graphics programs using the pop-up menu routines.

The *title bar* is the strip across the top of each window that always lets you access the *mex* window functions. Requesting a menu while the cursor is over the title bar lets you change the shape or appearance of the window, no matter where the input focus is.

1.3 What Does *mex* Do?

mex is the IRIS's window manager. With the *mex* menus, you can

- redirect input focus to a specific window
- change which window is on top, when two or more are overlapping
- move windows
- reshape windows
- kill windows

There is also a procedural interface to the window manager. Programmers can use Graphics Library routines to write a program that uses features of the window manager. When you program, you can use the *mex* routines to do the following:

- change the order in which the windows appear on the screen
- reshape windows
- move the input focus
- create pop-up menus

The *mex* routines are explained throughout this document.

1.4 Creating Windows

You create a new window that runs a new UNIX shell by placing the cursor outside any windows on your screen; then press and hold the right mouse button. The background menu appears (Figure 1-1):

Figure 1-1. Background Menu

Position the cursor over the 'new shell' item in the background menu and release the right mouse button. The cursor now appears as a red right angle (which represents the upper left corner of a window). To sweep out the window, position the cursor where you want to place the upper-left corner of the new window. Press and hold the right mouse button and drag the cursor diagonally across the screen until the red window outline is the desired shape and size. Release the right mouse button and the new window appears.

1.5 Interacting with the Window Manager

You can interact with the window manager by using the mouse in two different ways:

- select items from the title bar menu
- move the input focus directly to the window manager using the background menu

The title bar menu appears when you move the cursor over the title bar, then press and hold the right mouse button (see Figure 1-2). To select an item from this menu, move the cursor over the item so the item becomes highlighted, then release the mouse button. The title bar menu also appears when the cursor is on a window border or within a window, and you press and hold the right mouse button. This menu allows you to manipulate the window(s) on your screen. The items on the title bar menu are explained in the sections below.

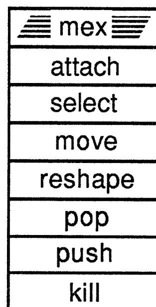


Figure 1-2. Title Bar Menu

The background menu (Figure 1-1) appears when the cursor is over the screen background, not touching any window, and you press and hold the right mouse button. The 'attach' item on the background menu attaches the input focus to the window manager. See Section 1.7, Attaching to the Window Manager.

1.5.1 Attaching to Windows

You attach to a window by positioning the cursor over the 'attach' item in the title bar menu, then releasing the right mouse button. You have successfully attached to a window when there is a red border around it. When you attach to a window, the mouse button events and keyboard input are directed to it, i.e., it receives the input focus.

1.5.2 Selecting a Window

You select a window by positioning the cursor over the 'select' item in the title bar menu, then releasing the right mouse button. Selecting a window simultaneously directs the input focus to that window and pops it to the top of the window stack. You have successfully selected a window when it appears on the top of the window stack and has a red border around it.

1.5.3 Moving a Window

You move a window by positioning the cursor over the 'move' item in the title bar menu, then releasing the right mouse button. When you do this, a different cursor appears (four small arrows, each pointing in a different direction). As you move this cursor, the red window outline moves with it. Place the window in the desired position using the mouse, then press and release the right mouse button.

1.5.4 Reshaping a Window

You reshape a window by positioning the cursor over the 'reshape' item in the title bar menu, then releasing the right mouse button. When you do this, a different cursor appears (a red right angle, which represents the upper-left corner of the window). To reshape the window, position the cursor where you want to place the upper-left corner of the window. To sweep out the window, press and hold the right mouse button, drag the cursor diagonally across the screen until the red window outline is the desired shape, then release the right mouse button. The window appears in its new size and shape.

1.5.5 Pushing and Popping Windows

You pop or push windows by positioning the cursor over either the 'pop' or 'push' item in the title bar menu, then releasing the right mouse button. When you 'pop' a window, it places the window at the top of the stack. When you 'push' a window, it places it at the bottom of the stack.

1.5.6 Removing a Window

You remove a window by positioning the cursor over the 'kill' item in the title bar menu, then releasing the right mouse button. (You cannot remove the console window.) You see this confirmation menu:

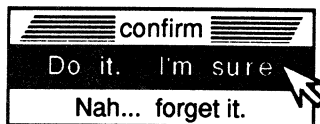


Figure 1-3. Confirmation Menu

Select 'Do it. I'm sure'.

1.6 Selecting a Title Font

You can specify the font that appears in the title bar by calling *mex* with the –t flag. *mex* then uses this font in the title bars of windows. Font files are located in */usr/lib/gl/fonts*.

```
mex  -t  /usr/lib/gl/fonts/<filename>
```

1.7 Attaching to the Window Manager

You can move the input focus to the window manger by positioning the cursor over the screen background, pressing and holding the right mouse button, then selecting 'attach' from the background menu (see Figure 1-1). The background menu appears only when you press and hold the right mouse button outside a window. Attaching to the window manager allows you to reshape, move, and pop windows without using the title bar menu. When you select 'attach' the cursor changes to an arrow that is inside three nested windows.

When you attach to *mex*, the mouse position and the mouse button events all go to the window manager. There are three possible mouse button events:

- Position the cursor over a window, then press the middle mouse button to move a window.
- Position the cursor near a corner of a window, then press the middle mouse button to reshape it.
- Position the cursor over a window, then press the left mouse button to pop it to the top of other windows.

1.8 Executing Graphics Programs from *mex*

To execute and open graphics programs that run under the window manager, follow these steps:

1. Type the name of the desired graphics program.
2. If a window does not appear immediately, examine the cursor shape.
3. If the cursor looks like four arrows and a red border is visible, move the window into the desired position. If the cursor looks like a right angle, sweep open the window.

2. Programming with *mex*

This chapter explains how to create and manipulate graphics windows from an application program. Sections 2.1 through 2.4 describe the Graphics Library routines, which are the building blocks for programming with *mex*. Section 2.5 gives general hints for window manager programming. Sections 2.6 and 2.7 contain sample programs: one that uses single buffer mode and one that uses double buffer mode. This chapter introduces the basics of window control. Two advanced topics are covered in Chapter 3, Making Pop-up Menus, and Chapter 5, Controlling Multiple Windows from a Single Process.

The directory */usr/people/gifts/mextools* contains many useful tools. See *Graphics Library User's Guide*, Appendix E, Window Manager Programs, for a description of the material in this directory.

A *graphics window* is a window in which an application program draws an image. A *text window* runs a UNIX shell. You create graphics windows with the Graphics Library routines described in this chapter.

2.1 Opening and Closing Windows

There are two Graphics Library routines that open and close windows.

winopen

`winopen` creates a graphics window. You call `winopen` from within a program.

When the window manager opens the window, the cursor appears as a red angle.

Press and hold the right mouse button to locate one corner of the window, drag the cursor diagonally, then release the button to locate the opposite corner.

`winopen` performs these operations:

- It invokes `gbegin` (see the *Reference Guide*, `gbegin`).
- It returns a unique integer identifier called the *gid* (graphics window identifier) for each graphics window. You can use this identifier to keep track of windows (see Chapter 5).
- It puts the *calling process* (the process that opens the window) in the background. To keep the process in the foreground, see Section 2.2.
- It queues INPUTCHANGE (see Section 2.5) and REDRAW (see Chapter 5).
- It sets the viewport to the window that the user opens (see *Programming Guide*, Chapter 4).
- It sets the lower-left corner of the window to (0,0). These are called *window coordinates*.

The argument to `winopen`, *name*, indexes the file */etc/gl/deskconfig* described in Section 4.6.

```
long winopen(name)
char name[];

integer*4 function winope(name,length)
character*(*) name
integer*4 length

function winopen(name: pstring128):longint;
```

getport

`getport` also creates a graphics window, but its functionality has been replaced by `winopen`. Although programs with `getport` still run, you should use `winopen`. `getport` does not return a window identifier, as does `winopen`. When you call `getport`, the pseudo devices INPUTCHANGE and REDRAW are automatically queued.

```
getport(name)
char name[];

subroutine getpor(name, length)
character*(*) name
integer*4 length

procedure getport(name: pstring128);
```

winclose

`winclose` removes a window.

```
winclose(gid)
long gid;

subroutine winclo(gid)
integer*4 gid

procedure winclose(gid: longint);
```

2.2 Setting Window Constraints

The window manager allows you to control the size, location, and shape of graphics windows from an application program. Calling `winopen` without first specifying any of these characteristics allows users to open a window of any size or shape anywhere on the screen. But if you'd like something specific, such as a small square window with a border, you must specify the desired size and shape.

Use the window constraint routines (see Table 2-1) to specify window characteristics. Call these routines before opening a window. Then, when you call `winopen`, *mex* applies those specifications. To specify characteristics on an existing window, see `winconstraints`, below.

When you interactively change a window, *mex* applies the constraints that you specify. Interactively changing a window means sweeping out a window with the mouse, or using *mex* pop-up menus to change the size, shape, or position of a window. To change windows noninteractively (that is, from an application program), see Section 2.3.

To specify	Use
Minimum size	`minsize`
Maximum size	`maxsize`
Aspect ratio	`keepaspect`
Size, in pixels	`prefsize`
Size and location	`prefposition`
Units for sizing	`stepunit`
Small increase in size	`fudge`
Process that draws background	`imakebackground*`
No screen space needed	`noport*`
Program runs in foreground	`foreground*`

* Call these routines before opening a window. These routines do not work with `winconstraints`.

Table 2-1. Window Constraint Routines

minsize

`minsize` specifies a minimum size for the graphics window. You cannot interactively reshape the graphics window to be smaller than this minimum size. The default minimum size is 40 pixels wide by 30 pixels high.

```
minsize(x, y)
long x, y;

subroutine minsiz(x, y)
integer*4 x, y

procedure minsize(x, y: longint);
```

maxsize

`maxsize` specifies a maximum size for the graphics window. The default maximum size is 1024 pixels wide by 768 pixels high.

```
maxsize(x, y)
long x, y;

subroutine maxsiz(x, y)
integer*4 x, y

procedure maxsize(x, y: longint);
```

keepaspect

`keepaspect` specifies the aspect ratio (width-to-height ratio) of the graphics window. You can resize the graphics window, but its shape stays the same. Use `keepaspect(1,1)` to ensure that the graphics window is always square. Use `keepaspect(4,3)` to ensure that the graphics window maintains the ratio four units wide to three units high, no matter how large or small the user resizes the window to be, or the size constraints imposed by calls to other window constraint routines.

```
keepaspect(x, y)
long x, y;

subroutine keepas(x, y)
integer*4 x, y

procedure keepaspect(x, y: longint);
```

prefsize

`prefsize` specifies the size of the graphics window as *x* pixels by *y* pixels. When `prefsize` is in effect, you cannot interactively resize the graphics window.

```
prefsize(x, y)
long x, y;

subroutine prefsi(x, y)
integer*4 x, y

procedure prefsize(x, y: longint);
```

prefposition

`prefposition` specifies the location and size of the graphics window. When `prefposition` is in effect, you cannot interactively relocate or resize the window.

```
prefposition(x1, x2, y1, y2)
long x1, x2, y1, y2;

subroutine prefpo(x1, x2, y1, y2)
integer*4 x1, x2, y1, y2

procedure prefposition(x1, x2, y1, y2: longint);
```

stepunit

`stepunit` makes the graphics window change size in steps of *xunit* and *yunit*. For example, an *xunit* of 5 makes it convenient to draw five columns of the same width in the graphics window, since the width of the graphics window is always a multiple of 5.

```
stepunit(xunit, yunit)
long xunit, yunit;

subroutine stepun(xunit, yunit)
integer*4 xunit, yunit

procedure stepunit(xunit, yunit: longint);
```

fudge

`fudge` adjusts the size of a graphics window so that a program can draw a border or add a heading. You can use `fudge` and `stepunit` together. For example, an *xunit* of 5 and an *xfudge* of 2 make it convenient to create five equal columns with a small border around them.

```
fudge(xfudge, yfudge)
long xfudge, yfudge;

subroutine fudge(xfudge, yfudge)
integer*4 xfudge, yfudge

procedure fudge(xfudge, yfudge: longint);
```

imakebackground

`imakebackground` sets up a process that draws the background of the window manager's screen. Use this routine for creating nonstandard (for example, patterned) backgrounds. The process must first draw the background and then read the event queue. The process redraws the background for every REDRAW token in the event queue (see Section 2.5). A program that declares itself as `imakebackground` stops any previous `imakebackground` program.

```
imakebackground()

subroutine imakeb

procedure imakebackground;
```

noport

`noport` tells the window manager that the program requires no screen space. Programs that only read or write the color map use this routine. After a call to `noport`, *mex* opens no additional windows for that process.

```
noport()

subroutine noport

procedure noport;
```

foreground

`foreground` makes a program run in the foreground. When you call `foreground` before opening the first window in a program with `winopen`, the window manager runs your graphics program as a background process. When a program runs in the background, it doesn't normally recognize input from the keyboard. However, a background program can be made to recognize keyboard and other input when both of the following conditions are met.

- You attach input to a window created by the background program.
- You queue the devices that you want the program to recognize.

```
foreground()

subroutine foregr

procedure foreground;
```

2.2.1 Setting Constraints for Existing Windows

You can call any of the ten window constraint routines just before you call `winopen`. *mex* applies these constraints when it opens the window. To respecify constraints for a window that is already open, follow these steps:

1. Call the desired series of window constraint routines (see Table 2-1).
2. Call `winconstraints`.

winconstraints

`winconstraints` restricts the size, location, or shape of a window as specified by the first seven routines in Table 2-1. `noport`, `imakebackground`, and `foreground` do not work with `winconstraints`.

```
winconstraints()

subroutine wincon

procedure winconstraints;
```

The code sample below removes all old constraints and subjects future resizing of the window to a 400x400 pixel minimum, with a square aspect ratio.

```
minsize(400, 400);
keepaspect (1, 1);
winconstraints();
```

When you call `winconstraints`, the appearance of the window doesn't immediately change. To put the changes into effect, you must interactively reshape the window with the mouse or the *mex* pop-up menus. `winconstraints` uses the constraints that were set the last time you created a window or called `winconstraints`. To remove all previous constraints, call `winconstraints` without any new constraints.

```
winconstraints();
```

2.3 Changing Windows Noninteractively (from within a Program)

This section describes how to give a program the same control over windows that you have when using the pop-up menus shown in Chapter 1, Getting Started with *mex*. You can move, resize, push, and pop windows from your program. You can also attach the input focus to a window, so that the input from all devices (mouse, keyboard, etc.) is directed to it. Table 2-2 lists the routines that perform these tasks.

Task	Routine
Move and reshape the current graphics window	`winposition`
Move lower-left corner of the current graphics window	`winmove`
Push the current graphics window	`winpush`
Pop the current graphics window	`winpop`
Attach input focus to the calling process and its current graphics window	`winattach`

Table 2-2. Window Control Routines

The window manager performs the operations in Table 2-2 on the current graphics window, which you can set with `winset` (see Section 2.4, Other Window Routines).

winposition

`winposition` moves and reshapes the current graphics window to the screen coordinates *(x1, y1)* and *(x2, y2)*. You can specify any two corners with the two *x* and two *y* coordinates. *mex* automatically erases the window at the old position.

```
winposition(x1, x2, y1, y2)
long x1, x2, y1, y2;

subroutine winpos(x1, x2, y1, y2)
integer*4 x1, x2, y1, y2

procedure winposition(x1, x2, y1, y2: longint);
```

winmove

`winmove` moves the lower-left corner (origin) of the current graphics window to screen location *(x, y)*. The size and shape of the window don't change. *mex* automatically erases the window at the old position.

```
winmove(orgx, orgy)
long orgx, orgy;

subroutine winmov
integer*4 orgx, orgy

procedure winmove(var orgx, orgy: longint);
```

`winposition` and `winmove` act on the current graphics window despite any constraints you have set. The window constraint routines (see Table 2-1) affect only the *interactive* reshaping or moving of windows. When you use `winmove` to move a window, it doesn't have to stay in the same position forever. Later, if you interactively move it (i.e., by using *mex* pop-up menus), it is subject to the constraints that were in effect at the last call to `winconstraints`.

winpush

`winpush` pushes the current graphics window. When you push a window, all overlapping windows obscure it.

```
winpush()

subroutine winpus()

procedure winpush;
```

winpop

`winpop` pops the current graphics window. When you pop a window, it covers all the windows that occupy the same portion of the screen.

```
winpop()

subroutine winpop

procedure winpop;
```

winattach

`winattach` moves the input focus to the current process. This routine corresponds to the 'attach' entry in the *mex* pop-up menu. `winattach` returns the identifier of the window that most recently had input focus.

```
long winattach()

integer*4 function winatt()

function winattach: longint;
```

2.4 Other Window Routines

This section describes the graphics window routines shown in Table 2-3.

Task	Routine
Make a title bar for the current graphics window	`wintitle`
Return the size of a graphics window	`getsize`
Return the origin of the graphics window	`getorigin`
Put the program in screen space	`screenspace`
Put viewport at current graphics window position	`reshapeviewport`
Enable the entire screen for writing	`fullscrn`
End full-screen mode	`endfullscrn`
Return the status of window manager	`ismex`
Return identifier of current graphics window	`winget`
Set the current graphics window	`winset`
Return identifier of graphics window beneath cursor	`winat`

Table 2-3. Miscellaneous Window Routines

wintitle

`wintitle` makes a title bar for the current graphics window, which you can set with `winset` or `winopen`. To assign colors to a title bar, see Chapter 4, Customizing *mex*, Section 4.3.

Calling `wintitle("")` removes the title bar of the current graphics window. The *name* argument is displayed on the left-hand side of the title bar. The extra FORTRAN argument, *length*, is the number of characters in the name string for the title bar.

```
wintitle(name)
char name[];

subroutine wintit(name,length)
character*(*) name
integer*4 length

procedure wintitle(name: pstring128);
```

getsize

`getsize` returns the size of the graphics window.

```
getsize(x, y)
long *x, *y;

subroutine getsiz(x, y)
integer*4 x, y

procedure getsize(var x, y: longint);
```

getorigin

`getorigin` returns the position of the origin (lower-left corner) of the graphics window.

```
getorigin(x, y)
long *x, *y;

subroutine getori(x, y)
integer*4 x, y

procedure getorigin(var x, y: longint);
```

screenspace

`screenspace` puts the window in screen space. Graphics positions are expressed in absolute screen coordinates. (Normally, when you open a window, (0,0) is the lower-left corner of the window, rather than the lower-left corner of the screen.) In screen space, (0,0) is the lower-left corner of the screen; (1279,1023) is the upper-right corner of the screen. Putting the program in screen space allows you to read pixels and locations outside the graphics window.

```
screenspace()

subroutine screen

procedure screenspace;
```

reshapeviewport

`reshapeviewport` sets the viewport to the current dimensions of the graphics window.

```
reshapeviewport()

subroutine reshap

procedure reshapeviewport;
```

fullscrn

`fullscrn` enables the entire screen for writing. It sets the screenmask and the viewport to the entire screen, with respect to the physical screen origin, not the origin of the graphics window. When you use this routine, graphics processes are not protected against interference from each other.

```
fullscrn()

subroutine fullsc()

procedure fullscrn;
```

endfullscrn

`endfullscrn` ends full screen mode. The screenmask and viewport are reset to the boundaries of the current graphics window. The current transformation is unchanged (see *Programming Guide*, Chapter 4).

```
endfullscrn()

subroutine endful

procedure endfullscrn;
```

ismex

`ismex` returns TRUE(1) if the window manager is running and FALSE(0) otherwise.

```
Boolean ismex()
```

```
logical function ismex()
```

```
function ismex(): Boolean;
```

winget

`winget` returns the graphics window identifier (*gid*) of the current graphics window.

```
long winget()
```

```
integer*4 function winget()
```

```
function winget(): longint;
```

winset

`winset` makes the specified graphics window current. When you have multiple windows controlled from a single process, you must specify the window into which you want to direct the graphics.

```
winset(gid)
long gid;
```

```
subroutine winset(gid)
integer*4 gid
```

```
function winset(gfnum; longint): longint;
```

winat

`winat` returns the graphics window identifier (gid) of the window beneath the current position of the cursor.

```
long winat()

integer*4 function winat()

function winat(): longint;
```

2.5 Programming Hints

When you program under the window manager, you should be aware of the special considerations listed in the sections below.

2.5.1 Graphics Initialization

`winopen` initializes graphics and replaces `ginit`.

2.5.2 Shared Facilities

All windows on the screen share a limited number of cross-bar mappings. Within each mapping, colors are shared. Graphics programs that run at the same time must cooperate in their color map usage. Windows created by the same program share patterns, raster fonts, and cursor glyphs.

2.5.3 Raster Fonts

Raster fonts are not scaled. The size of graphics windows and of most graphical images can vary, but the raster font text stays the same size.

2.5.4 The Event Queue

Each graphics program has an event queue (see *Programming Guide*, Chapter 7). REDRAW and INPUTCHANGE contain the value of the *gid* (graphics window identifier) of the window that is attached, or zero if the input focus is removed. A REDRAW token appears in the queue under these circumstances:

- *mex* moves or reshapes a window.
- Part of a window is uncovered.
- The display mode changes.

Programs can also queue the pseudo-device INPUTCHANGE. This pseudo-device indicates a change of state in the system.

INPUTCHANGE indicates whether a user has attached to or detached from a window.

2.6 Sample Program: Single Buffer Mode

This section contains a program that runs under the window manager in single buffer mode. `keepaspect`, `winopen`, and `reshapeviewport` are the only window manager routines needed to run this program under the window manager.

The *drawit* subroutine, which makes a spiral out of circles, executes once. Then, whenever the user interactively moves the window, *drawit* executes again. `reshapeviewport` makes sure the viewport is set to the new location of the current graphics window.

■ C Program: SINGLE BUFFER MODE

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

main()
{
    short val;
```

```
    keepaspect(1,1);  /* the graphics window can be at any
                    location and size, as long as it's square */
    winopen("zoing");
    drawit();       /* image drawn the first time */
        /* the image is redrawn whenever a REDRAW
            appears in the event queue */
    while(1) {
    if(qread(&val) == REDRAW)
        drawit();
    }
}

drawit()
{
    register int i;

    reshapeviewport();
    color(7);
    clear();
    ortho2(-1.0,1.0,-1.0,1.0);
    color(0);
    translate(-0.1,0.0,0.0);
    pushmatrix();
    for(i=0; i<200; i++)   {
        rotate(170,'z');
        scale(0.96,0.96,0.0);
        pushmatrix();
        translate(0.10,0.0,0.0);
        circ(0.0,0.0,1.0);
        popmatrix();
    }
    popmatrix();
}
```

2.7 Sample Program: Double Buffer Mode

This double buffered program draws a cube that the user rotates by moving the mouse.

■ C Program: DOUBLE BUFFER MODE

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

main()
{
    int x, y;       /*  current rotation of object    */
    short active;       /*  TRUE if window is attached*/
    short dev, val;

    keepaspect(3,2);
    winopen("cube");
    doublebuffer();
    gconfig();
    qdevice(INPUTCHANGE);
    qdevice(REDRAW);
    qdevice(ESCKEY);
    qdevice(MOUSEX);
    qdevice(MOUSEY);
    perspective(400, 3.0/2.0, 0.001, 100000.0);
    translate(0.0, 0.0, -3.0);
    rotate(900, 'z');

    x = 0;  y = 0;
    active = 0;
    while(TRUE) {
    drawcube(x,y);      /*  draw into the back buffer */
    swapbuffers();      /*  show it in the front buffer */
    while (qtest()) {   /*  process queued tokens */
        dev = qread(&val);
        switch(dev) {
        case ESCKEY:    /*  exit program with ESC */
            exit(0);
            break;
        case INPUTCHANGE:
```

```
            active = val;
            break;
        case REDRAW:
            reshapeviewport();
            break;
        case MOUSEX:
            x = val;
            break;
        case MOUSEY:
            y = val;
            break;
        default:
            break;
        }
    }
    }
}

drawcube(rotx,roty)
int rotx, roty;
{
    color(0);
    clear();
    color(7);
    pushmatrix();
    rotate(rotx,'x');
    rotate(roty,'y');
    cube();
    scale(0.3,0.3,0.3);
    cube();
    popmatrix();
}

cube() /* make a cube out of 4 squares */
{
        pushmatrix();
                side();
                rotate(900,'x');
                side();
                rotate(900,'x');
                side();
```

```
                rotate(900,'x');
                side();
        popmatrix();
}

side() /* make a square translated 0.5 in the z direction */
{
        pushmatrix();
                translate(0.0,0.0,0.5);
                rect(-0.5,-0.5,0.5,0.5);
        popmatrix();
}
```

3. Making Pop-up Menus

This section describes routines you can put in your graphics programs to create and use pop-up menus. When you select an item from a menu, these routines automatically identify which menu item has been selected.

Sections 3.1 and 3.2 describe the routines that create and call up pop-up menus. Section 3.3 tells how to choose colors for pop-up menus. Section 3.4 tells how to change the return value of a menu item, bind functions to menus and items, and make title bars and rollover menus. Sections 3.5 and 3.6 provide examples of pop-up menu routines.

The pop-up menu routines work only when you are running the window manager. The window manager reserves some of the system image memory for overlays, pop-up menus, and cursors. When the cursor moves or a pop-up menu disappears, the system redisplays the underlying image, which has been saved. The system doesn't need to do any redrawing.

Table 3-1 correlates tasks with pop-up menu routines.

Task	Routine
Make a new menu	`newpup`
Add menu items	`addtopup`
Make a menu with items	`defpup`
Call up a menu	`dopup`
Delete a menu	`freepup`
Enable pop-up menu bitplanes for writing	`pupmode`
End pop-up mode	`endpupmode`

Table 3-1. Pop-up Menu Routines

3.1 Creating a Pop-up Menu

To create pop-up menus, you must be able to write into the two highest-order bitplanes. The window manager uses these bitplanes to display pop-up menus. An application program can access these bitplanes and retain input focus without conflicting with other processes. It is recommended that only processes with input focus write in these bitplanes. If you program in C, you can create a menu in two ways:

- Initialize the menu with `newpup`, and then add menu items with `addtopup`.
- Use `defpup`, which combines the functions of `newpup` and `addtopup`.

If you program in FORTRAN or Pascal, you must use `newpup` and `addtopup` to create your menu, since `defpup` is not available in FORTRAN or Pascal.

newpup

`newpup` allocates and initializes a structure for a new menu. This function takes no arguments and returns a 32-bit integer identifier for the pop-up menu.

```
long newpup()
```

```
integer*4 function newpup()
```

```
function newpup:longint;
```

defpup

`defpup` allocates structure and makes menu entries. From C, you can combine the functions of `newpup` and `addtopup` by calling `defpup`. (This routine is available only in C.)

```
long defpup(str [, args] ...)
char *str;
long args;
```

`defpup` creates the menu shown below in one step (see `addtopup`).

defpup creates the menu shown below (under (underaddtopupbelow in one step.

```
menu = defpup ("first|second|third");
```

You can add additional menu entries with addtopup.

addtopup

newpup creates an empty menu. To build a menu, use both newpup and addtopup. addtopup adds menu entries to the bottom of an existing menu.

pup is the menu identifier returned by newpup or defpup. *str* is a character string that specifies the entries in the menu. The string lists the menu labels from the top to the bottom of the menu, with a '|' (vertical bar) between entries.

```
addtopup(pup, str, arg)
long pup;
char *str;
long arg;

subroutine addtop(pup, str, length, arg)
integer*4 pup
character* str(*)
integer*4 length
integer*4 arg

procedure addtopup(pup: longint; str: pstring128;
        arg: longint);
```

The FORTRAN version of addpup takes an additional *length* argument, which specifies the number of characters in the string. *arg* is necessary only for advanced menu formats (see Section 3.4).

To create a menu that looks like this:

first
second
third

use these routines:

FORTRAN:

```
IMENU = NEWPUP()
CALL ADDTOP (IMENU, "first|second|third", 18)
```

The *18* in the FORTRAN routine is the number of characters in the string, including the vertical bars.

If you elect not to use the mex pop-up facility, use `pupmode` and `endpupmode` to gain access to high-order bitplanes for menu drawing.

pupmode

`pupmode` enables the two highest-order bitplanes for writing.

```
pupmode()

subroutine pupmod

procedure pupmode;
```

endpupmode

`endpupmode` disables writing to the pop-up menu bitplanes.

```
endpupmode()

subroutine endpup

procedure endpupmode;
```

3.2 Calling Up a Pop-up Menu

When the window manager is running, normally you cannot write outside window boundaries. But you might want your pop-up menus to appear there. In this case, use `fullscrn` to enable the entire screen for writing. `endfullscrn` ends full-screen mode. (See Chapter 2, Programming with *mex*, Section 2.4, for a more complete description of full-screen mode.)

To use pop-up menus created by application programs, make sure there is no *reservebut* entry for the right mouse button in your *.mexrc* (see Section 3.3 for an explanation).

dopup

dopup displays an existing pop-up menu.

pup is the identifier of the pop-up menu. The system displays the menu until the user either makes a selection or releases the button with the cursor off the menu. The value that dopup returns depends on the user's menu selection. If the user doesn't make a selection, dopup returns –1. If the pop-up menu does not use an advanced menu format (see Section 3.4), dopup returns an integer corresponding to the position of the item in the menu.

```
long dopup(pup)
long pup;

integer*4 function dopup(pup)
integer*4 pup

function dopup(pup: longint): longint;
```

To make the right mouse button to bring up the menu shown in Section 3.1, use this code:

C:

```
dev = qread(&val);
if (dev == RIGHTMOUSE) {
    if (val == 1) {        /*  right mouse button is pressed  */
        menuval = dopup (menu);
    }
}
```

FORTRAN:

```
IDEV = QREAD(IVAL)
IF (IDEV .EQ. RIGHTM) THEN
    IF (IVAL .EQ. 1) THEN
        IMVAL = DOPUP (IMENU);
    ENDIF
ENDIF
```

You select 'first', 'second', or 'third' by releasing the button with the cursor over that entry. You make no selection by releasing the button with the cursor off the menu. Table 3-2 shows the return value for each possible selection.

Selection	Return Value
first	1
second	2
third	3
no selection	–1

Table 3-2. Default Return Values

To get different values back, see Section 3.4.

freepup

`freepup` deletes a pop-up menu.

```
freepup(pup)
long pup;

subroutine freepu(pup)
integer*4 pup

procedure freepup(pup: longint);
```

3.3 Choosing Colors for Pop-up Menus

To choose colors for pop-up menus, include some special commands in your *.mexrc* file. A typical *.mexrc* contains these lines:

```
bindcolor  cursor  255  0  0
bindcolor  menu  0  0  0
bindcolor  menuback  255  255  255
```

The three numbers in each line stand for red, green, and blue values. These lines tell *mex* which colors to use for pop-up menus. *mex* uses the same colors for all pop-up menus in all programs. *menu* stands for the text of the menu, *menuback* for the background, and *cursor* for the cursor. In the example above, the color values R=G=B=0 for *menu* make the text of the menu black. The color values R=G=B=255 for *menuback* make the background white; R=255, G=0, B=0 for *cursor* make the cursor bright red.

pupcolor

`pupcolor` specifies the current color for drawing in the pop-up bitplanes. The possible colors range from 1 to 3, and are determined by *bindcolor* commands in your *.mexrc* file. 1 corresponds to the color bound to the cursor, 2 to the color bound to the menu, and 3 to the color bound to the menu background.

```
pupcolor(c)
long c;

subroutine pupcol(c)
integer*4 c

procedure pupcolor(c: longint);
```

Table 3-3 shows the correspondence between the arguments to `pupcolor`, the commands in your *.mexrc* file, and the define values in *gl.h*.

Pupcolor Argument	.mexrc Command	gl.h Define Value
0	no pop-up menu color	
1	bindcolor cursor	PUP_CURSOR
2	bindcolor menu	PUP_BLACK
3	bindcolor menuback	PUP_WHITE

Table 3-3. Colors for Drawing Pop-up Menus

Under the window manager, you have access to only one-fourth the colors available outside the window manager. The system reserves the other three-fourths of the indices in the color table (non-zero values in the first two bitplanes) for overlays, pop-up menus, and cursors. Users cannot overwrite these reserved color indices.

3.4 Advanced Menu Formats

You can use advanced menu formats to:

- change the value returned when you select a menu item
- bind a function to a whole menu or to a menu item
- make a title bar or a rollover menu

You introduce the special format when you call `defpup` or `addtopup`. After the string that defines the menu entry, add these format instructions:

- Percent sign (%) begins the special format.
- One format character specifies which format to use (see Table 3-4).
- Numeric values and/or arguments are required for some formats. Numeric values immediately follow the format character. Arguments always come at the end of the `defpup` or `addtopup` routine.

More than one special format can be associated with a menu entry. Use vertical bars to separate entries.

3.4.1 Getting Back the Default Values for Menu Selections

The %n format is the default; specifying this format is the same as not specifying any format (see Section 3.2). The %n format takes no arguments. The menu:

```
menu = newpup();
addtopup (menu, "first|second|third");
```

is the same as the menu:

```
menu = newpup();
addtopup (menu, "first %n|second %n|third %n");
```

3.4.2 Changing the Return Values for Menu Selections

The %x format changes the numeric value that dopup returns. If you define this menu:

```
menu = newpup();
addtopup (menu, "first %x15|second %x7");
```

selecting 'first' causes dopup to return 15, not 1. Selecting 'second' causes dopup to return 7, not 2. For this format, you must specify a numeric value that dopup returns in place of the default.

3.4.3 Making a Title Bar

The %t format creates a title bar on a pop-up menu. You can't select the title bar; it doesn't highlight.

```
menu = newpup();
addtopup (menu, "Cardinal %t|first|second|third");
```

is the same as the first example above, except there is a title bar at the top of the menu. The %t format takes no arguments.

3.4.4 Binding a Function to a Whole Menu

The %F format specifies a function that affects all values returned by all items in the menu. This format requires an argument: the function that affects all values that dopup returns.

```
menu = newpup();
addtopup (menu, "Cardinal %t %F|first|second %x10", funct);
```

Selecting ''first'' causes `dopup` to return *funct(1)*, instead of 1. Selecting ''second'' causes `dopup` to return *funct(10)*.

3.4.5 Binding a Function to a Menu Entry

The %f format makes a menu entry that calls a function. The name of the function is the argument to this format.

```
menu = newpup();
addtopup (menu, "first|call %f", funct);
```

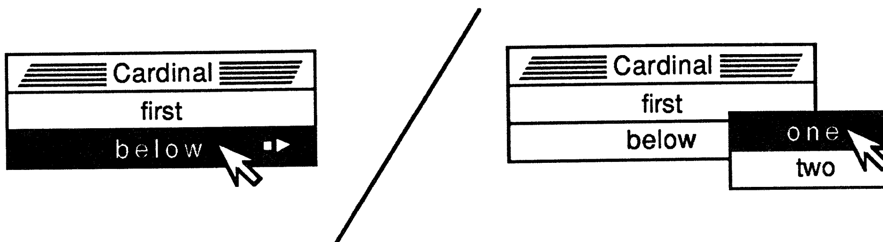
Selecting ''call'' executes the function *funct* with an argument of 2 (the default return value for this selection). `dopup(menu)` returns *funct(2)*.

3.4.6 Making a Nested (Rollover) Menu

The %m format creates a simple nested pop-up menu (a submenu item). When you roll the cursor off either the left or the right of the menu label, you invoke the submenu. Labels on the submenu can have further choices. This format requires an argument: the menu identifier for the pop-up submenu. This code:

```
submenu = newpup();
addtopup (submenu, "one|two");
menu = newpup();
addtopup (menu, "Cardinal %t|first|below %m", submenu);
```

creates these menus:



If you select an item from the submenu, `dopup(menu)` returns the same value as `dopup(submenu)`. If you display the submenu but don't select anything from it, dopup returns –1.

Table 3-4 summarizes the formats described in this section.

Task	Format	Changes return value?	Needs numeric value?	Needs argument?
Return default values	%n	no	no	no
Return other values	%x	yes	yes	no
Make title	%t	no	no	no
Bind function/whole menu	%F	yes	possibly	yes
Bind function/menu item	%f	yes	possibly	yes
Make nested menu	%m	yes	no	yes

Table 3-4. Summary of Advanced Menu Formats

3.5 An Example from *cedit*, a Color Editing Program

cedit is a simple color editor that runs under the window manager. The source code for the program and the executable file are both available on your IRIS in */usr/people/gifts/mextools/tools.*

The program brings up a window containing a color control system in which red, green, and blue sliding bars change the color components of a sample patch of color. The right mouse button brings up a menu that allows you to choose from four color systems; the left mouse button selects a color on the screen to edit and also controls the sliding bars.

The *cedit* window is created by three lines of code:

```
keepaspect(1,1);
winopen("cedit");
menu = defpup("colorsys %t|rgb|cmy|hsv|hls");
```

`keepaspect`, one of the window constraint routines discussed in Chapter 2, Section 2.2, requests a square aspect ratio. You must call `keepaspect` before `winopen` so that the window manager enforces this constraint when it creates the window. The pop-up menu definition creates a menu with title 'colorsys' and menu entries 'rgb', 'cmy', 'hsv', and 'hls'.

The code sample below sets the color system according to your menu selection.

```
dev = qread(&val);
switch (dev) {
        case MENUBUTTON:
                sel = dopup(menu);
                if (sel>0) {
                    setcolorsys(sel);
                    newcolor(cc);
                }
                break;
```

When you make a selection from the pop-up menu by clicking the right mouse button, `dopup(menu)` returns a value. Since the pop-up menu definition does not include any advanced menu formats, the return value is the default: 1 for the first selection ('rgb'), 2 for the second ('cmy'), and so on. The return value is –1 if you don't make a selection.

The program then assigns the return value to the variable *sel*. If *sel* is greater than zero (i.e., if you make a selection), then the program sets the color system accordingly, and calls *newcolor*, a routine defined in the *cedit* program. *newcolor* sets the color of the sample patch, and sets the sliding bars to the correct position for that color within the selected color system.

The case statement started in the code sample above continues with instructions for the middle and left mouse buttons. If you point to a color outside the *cedit* window, a click of the middle mouse button changes that color to the color of the sample patch. A click of the left mouse button retrieves one pixel and makes the color of that pixel the current color for editing.

If you know which color system you want to use for editing a color, you can bypass the 'colorsys' menu by giving *cedit* an argument. The argument corresponds to the position of the color system in the 'colorsys' menu. The *cedit* window then appears with the color system already set. The code that makes this possible (shown below) immediately precedes the window

constraint and `winopen` routines.

```
main(argc,argv)
int argc;
char **argv;
{
    int i, j;

    if (argc>1)
        setcolorsys(atoi(argv[1]));
```

Not all the subroutines called by *cedit* are local to the program. The C code for the routine that retrieves a pixel for editing is currently available in */usr/people/gifts/mextools/portlib/getapixel.c*. The C code for *setcolorsys*, the routine that sets the color system, is currently available in */usr/people/gifts/mextools/portlib/colormod.c*.

In the same directory, the library *portlib.a* contains the object code for *setcolorsys*. When you execute *make* in *cedit*'s directory, this library is linked automatically. You can make these routines available to your programs by referencing */usr/people/gifts/mextools/portlib/portlib.a* (see */usr/people/gifts/mextools/README*).

3.6 Sample Program

This program demonstrates the use of the pop-up menu routines described in this chapter. The left mouse button puts shapes on the screen if you attach to the window entitled "blop".

■ C Program: BLOP

```
#include "gl.h"
#include "device.h"
#define CIRCLE 1
#define RECT   2
#define LINE   3
int mainmenu;
int colormenu;
int shapemenu;
int curcolor;
int curshape;
```

```
int xorg, yorg;

setshape(n)
int n;
{
    curshape = n;
    return -1;     /* dopup will return this value */
}

setcolor(n)
int n;
{
    curcolor = n;
    return 7;      /* dopup will return this value */
}

main()
{
    short val;
    int x, y;
    int pupval;

    prefsize(400,300);
    winopen("blop");
    wintitle("blop");
    curcolor = WHITE;
    curshape = CIRCLE;
    qdevice(LEFTMOUSE);
    qdevice(MENUBUTTON);
    shapemenu = defpup("shapes %t %F|Circle|Rect|Line",setshape);
    colormenu = defpup("colors %t %F|BLUE %x4|WHITE %x7|RED %x1",
        setcolor);
    mainmenu = defpup("blop %t|shapes %m|color %m|clear|set",
                  shapemenu,colormenu);
    makeframe();
    while(1) {
    switch(qread(&val)) {
        case REDRAW:
        makeframe();
        break;
        case LEFTMOUSE:
```

```
        if(val) {
            x = getvaluator(MOUSEX)-xorg;
            y = getvaluator(MOUSEY)-yorg;
            drawshape(curcolor,curshape,x,y);
        }
        break;
        case MENUBUTTON:
        if(val) {
            pupval=dopup(mainmenu);
            printf("dopup returns %d0,pupval);
            switch(pupval) {
            case 3:         /* clear */
                color(BLACK);
                clear();
                break;
            case 4:         /* set */
                color(WHITE);
                clear();
                break;
            }
        }
    }
    }
}

drawshape(acolor,ashape,x,y)
int acolor, ashape, x, y;
{
    color(acolor);
    switch(ashape) {
    case CIRCLE:
        circfi(x,y,10);
        break;
    case RECT:
        rectfi(x,y,x+15,y+10);
        break;
    case LINE:
        move2i(x,y);
        draw2i(x+20,y+20);
        break;
    }
```

```
}
makeframe()
{
    reshapeviewport();
    getorigin(&xorg,&yorg);
    color(BLACK);
    clear();
}
```

4. Customizing *mex*

This chapter explains how to change the default configuration of the window manager. Section 4.1 explains how the window manager interprets button events. Sections 4.2 and 4.3 tell how to bind color indices to the title bar and borders of a window and to pop-up menus and the cursor. Section 4.4 tells how to change a color map entry to a specified RGB value. Section 4.5 shows a sample *.mexrc* configuration file that is different from the default. Section 4.6 tells how to arrange your desktop. (You can think of the presentation of the full screen as a desktop.)

When you start the window manager, it tries to read a file called *.mexrc* in the directory you were in when you started *mex*. If it doesn't find one, it looks for *.mexrc* in your home directory. If you don't have a *.mexrc*, the window manager uses */usr/lib/gl2/mexrc*, which is shown in Figure 4-1. The entries you can put in a *.mexrc* configuration file are shown in Table 4-1.

In this chapter, the *default configuration* of the window manager refers to the *.mexrc* shown below. To change the defaults, copy */usr/lib/gl2/mexrc* to *.mexrc* in your home directory. Make changes to the file in your home directory. If you later decide to return to the default configuration, remove the changed file and restart *mex*.

```
#include <device.h>
bindfunc pop LEFTMOUSE
bindfunc movegrow MIDDLEMOUSE
bindfunc menu RIGHTMOUSE
bindindex inborder 7
bindindex hiinborder 7
bindindex titletextin 4
bindindex titletextout 4
bindindex hititletextin 4
bindindex hititletextout 4
bindindex hioutborder 1
bindindex outborder 0
bindcolor menu 10 10 10
bindcolor menuback 220 220 220
bindcolor cursor 255 0 0
```

Figure 4-1. Default *.mexrc* Configuration File

Task	Command
Reserve a button	`reservebut`
Bind a function to a button	`bindfunc`
Bind color index to window element	`bindindex`
Bind RGB values to reserved bitplanes	`bindcolor`
Change color map entry	`mapcolor`

Table 4-1. *.mexrc* Configuration File Entries

4.1 Interpreting Button Events

When you reserve a mouse or keyboard button, the window manager process always recognizes input from that button. In the default configuration, none of the mouse buttons is reserved, since this would make your applications unable to recognize mouse button input.

The window manager also allows you to associate buttons with actions. *bindfunc* binds a window manager function to a button. Table 4-2 gives the names of the bindable functions and the action that occurs when you press the bound button. See *IRIS User's Guide*, Appendix A, Type Definitions, for button numbers.

Function	Action
menu	Displays a pop-up menu. If the cursor is over a window, *mex* displays the title bar menu. If the cursor is not over a window, *mex* displays the background menu (see Chapter 1, Section 1.4).
hogmode	Makes *mex* the active process. All button events go to it until you attach to another window. **Note:** Binding this function to a button works only if you have also reserved the button with `reservebut`.
hogwhiledown	Sends all button events to *mex* when this button is down. When you release the button, the button events go to the window that was active most previously, unless you attached to another window when the button was down. **Note:** Binding this function to a button works only if you have also reserved the button with `reservebut`.
movegrow	Moves or resizes the window under the cursor, depending on the cursor position. If the cursor is near the center of a window, *mex* moves the window when the button is down and positions it when the button is released. If the cursor is near a corner of the window, *mex* moves that corner. **Note:** The window constraint routines are applied when you use *movegrow* (see Chapter 2, Programming with *mex*).
kill	Removes the window that is under the cursor. **Note:** You cannot kill the console window.
move	Moves the window that is under the cursor.
push	Pushes the window that is under the cursor.
pop	Pops the window that is under the cursor.
attach	Attaches to the window that is under the cursor.
popattach	Selects (pops and attaches to) the window that is under the cursor.

Table 4-2. Bindable Functions

The right mouse button is special in that it is the *menu button*; the *menu* function is bound to it. The binding of the *menu* function to the right mouse button is necessary for the *mex* pop-up menus to work correctly. You can change the bindings of the left and middle mouse buttons. Do not change the binding of the right mouse button.

In the default configuration, the *pop* function is bound to the left mouse button. If you are attached to the window manager process (i.e., to the screen background; see Chapter 1, Section 1.5), this button pops windows. The *movegrow* function is bound to the middle mouse button. If you are attached to the window manager process and the cursor is near the center of the window, the middle mouse button moves the window; if the cursor is near the corner of the window, the middle mouse button reshapes the window.

If you'd rather use the PF4 key to pop windows, you can edit your *.mexrc* to contain this line:

```
bindfunc pop 78
```

You need to restart *mex* to see the effect of your change. Then, when you are attached to the window manager process, the PF4 key pops windows.

In addition, you can reserve the PF4 key for the window manager's use, by editing your *.mexrc* to contain this line:

```
reservebut 78
```

With both of these entries in your *.mexrc*, the PF4 key pops windows even if you aren't attached to the window manager process. The disadvantage of this is that your application can't get input from the PF4 key (or any key reserved with `reservebut`), since it is reserved for the window manager.

In the default configuration, none of the mouse buttons are reserved; therefore, applications can recognize their input. Applications recognize right mouse button input when all of the following conditions are met.

- The program queues the right mouse button.
- The program's graphics window is active.
- The cursor is over the window.

However, if you press the right mouse button with the cursor over the window's border or title, the window manager process always recognizes the button event. The title bar menu appears. If you press the right mouse button with the cursor over the screen background, the window manager process also recognizes the button event. In this case, the background menu appears.

Applications recognize left and middle mouse button input if the program queues the buttons and if the program's graphics window is active. The cursor need not be over the window.

The window manager's recognition of button events depends on whether a window or the window manager is active. If a window is active, the window manager recognizes input only from reserved buttons. If the window manager itself is active, it recognizes all button events.

To make the window manager active, you can attach to the background, or you can bind the *hogmode* function to the button of your choice. You must reserve the button as well. Then, when you press the bound button, the *mex* process recognizes all button events. A variation of this function, *hogwhiledown*, causes *mex* to recognize button events when the bound button is down (see Table 4-2).

4.2 Binding Colors to the Title Bar and Borders

`bindindex` assigns color indices to the title bar and borders of a window. You can assign color indices to the regions shown in Figure 4-2.

inborder is the window's inner border and the background of the title bar. The inner border is one pixel wide. *outborder* is the window's outer border, which is also one pixel wide. *titletextin* is the title text; *titletextout* is the horizontal stripe in the window's title bar.

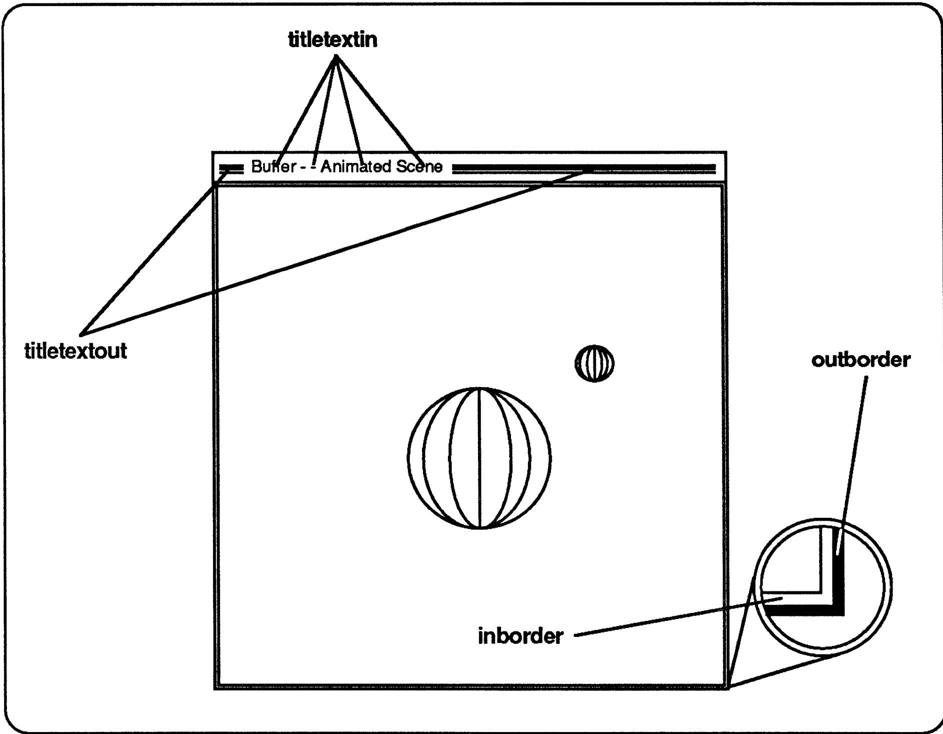


Figure 4-2. Title Bar and Border Regions of a Window

When you are attached to a window, its regions are highlighted. You can assign color indices to these highlighted regions: *hiinborder*, *hioutborder*, *hititletextin*, and *hititletextout.*

For example,

```
bindindex  hititletextin  7
```

binds color index 7 to the title text when the window is attached.

4.3 Binding Colors to Pop-up Menus and the Cursor

`bindcolor` binds RGB (red, green, and blue) colors to the window manager's reserved bitplanes, for the pop-up menu text, pop-up menu background, and cursor. For example,

```
bindcolor  cursor  0  0  255
```

makes the cursor blue;

```
bindcolor  menu  255  0  0
```

makes the menu text red;

```
bindcolor  menuback  0  255  0
```

makes the menu background green.

4.4 Setting Color Map Entries

`mapcolor` changes a color map entry to a specified RGB value. These lines set color map indices 1, 2, and 3 to red, green, and blue, respectively.

```
mapcolor  1  255  0  0
mapcolor  2  0  255  0
mapcolor  3  0  0  255
```

4.5 A Sample *.mexrc*

The configuration file shown below is an example of a *.mexrc* that is slightly different from the default shown in Figure 4-1. This configuration file frees all three mouse buttons for use in application programs. Use the no-scroll key (button 13) to issue a single window manager command, and the PF4 key (button 78) to issue a sequence of commands.

```
reservebut 78
bindfunc hogmode 78
reservebut 13
bindfunc hogwhiledown 13
bindfunc popattach 103
bindfunc movegrow 102
bindfunc menu 101
```

When you use this *.mexrc*, the window manager reserves these two buttons. When you press the PF4 key, all events go to the window manager, including all the mouse events. The right mouse button brings up the menu, the middle button invokes *movegrow*, and the left button invokes *popattach*. The window manager recognizes all keystrokes until you attach to a window.

While you hold down the no-scroll key, the window manager recognizes mouse events. When you release it, keystrokes go back to the window that was most recently active.

The configuration file above uses the default colors and color indices.

4.6 Arranging Your Desktop

When you log in, you may want to start *mex* and run some programs (for example, a clock program) automatically. To do this, follow these steps:

1. Make sure the programs under */usr/people/gifts/mextools* are already compiled; see the README in that directory.

2. Add entries to your *.login* file to check whether *mex* is running, start it if not, and run the programs. The following lines are sample lines from a *.login* file.

```
setenv PATH 'printenv PATH':/usr/people/gifts/mextools/tools
setenv PATH 'printenv PATH':/usr/people/gifts/mextools/imgto
ismex     # test to see whether mex is running
if ( $status == 0 ) then     # if not, then
     mex -d     # run mex in double buffer mode
     sleep 1     # wait a second
     makemap     # initialize the color map
     fade     # draw the background
     clock     # start up the clock
endif
```

3. Put entries in a file called *.deskconfig* in the directory where you start *mex*. If you start *mex* in your *.login* file, you should put *.deskconfig* in your home directory. The entries in *.deskconfig* tell *mex* where to place windows for each program you name.

   Each entry in *.deskconfig* appears on its own line, and has the form

   ```
   window: x1, y1, x2, y2
   ```

   window is the name of the graphics program to be positioned. *x1, y1* and *x2, y2* are two corners of the window. If *x1* is equal to *x2* and *y1* is equal to *y2*, the specified corner is the lower-left corner of the window.

   For example, to specify that when the clock program is run, it should always appear in the lower-right corner of your screen, add this line:

   ```
   clock: 950  35  1015  100
   ```

4. Exit from *mex* and log out.

5. Log in again. *mex* starts automatically with a clock in the lower-right corner.

5. Controlling Multiple Windows from a Single Process

This chapter describes functions that allow you to control multiple windows from a single process. The window manager associates a unique integer name, called the gid (graphics window identifier), with each window on the screen. This name allows you to keep track of windows.

If a process controls several windows, and the user uncovers (pops) one of them, the window manager sends the process a REDRAW token to redisplay the contents of the uncovered window. Along with the REDRAW token, the window manager sends the gid of the window, so that the process knows where to redisplay the image (see Figure 5-1). No two windows can have the same gid, even if the windows are created by different processes.

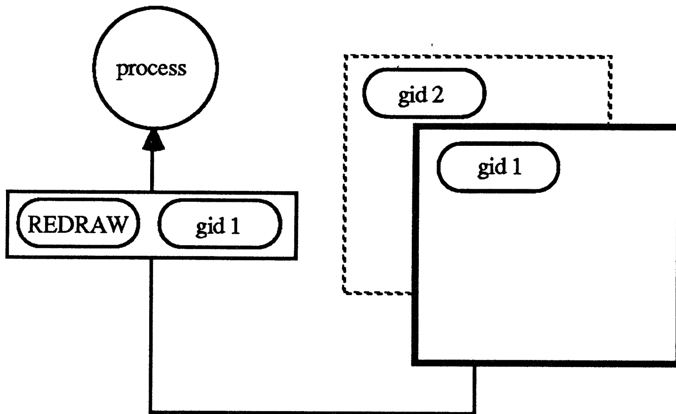


Figure 5-1. Processing of REDRAW Token and gid

The code segments in this section show how an an application program (a single process) can control multiple windows. The entire program is reprinted at the end of this chapter for reference.

To create the windows, call winopen:

```
for (i = 0; i < nwins; i++) {
        win_ids[i] = winopen ("cubes");
        if (i == 0) {
                doublebuffer();
                gconfig();
        }
        perspective (900, 1.0, 1.0, 3.5);
        translate (0.0, 0.0, -2.0);
}
```

You can create a maximum of nine windows from one program. To keep track of the windows, the win_ids array saves the gid for each window. Each newly opened window becomes the current graphics window.

The current graphics window is the window where drawing and window manipulation take place. Only one graphics window is current for any process.

To draw into a particular window, use winset to select a window as the current window. The program segment below sets each window as the current window, and then draws the cube into each window with a different scale factor.

```
for (i = 0; i < nwins; i++) {
        winset (win_ids[i]);
        pushmatrix ();
        cube (WHITE, cumex, cumey, cumez, win_ids[i]);
        popmatrix ();
}
```

Each window has its own matrix stack. You can modify the matrix stack for the current graphics window only.

The ability to control multiple windows from a single process does create some graphics management problems. For example, when you change (push, pop, or resize) a window, it is most efficient to update only that window, not all windows. When a window changes, the window manager sends the process both a REDRAW token and the gid. The window manager sends one REDRAW token for each window that needs updating. The code segment below shows how to use the gid to update only windows that have changed. Note that winset expects a *long* for the gid.

```
case REDRAW:
/*  only redraw windows that need to be redrawn  */
        winset ((long) val);
        reshapeviewport ();
        pushmatrix();
        frontbuffer(TRUE);
        cube (WHITE, cumex, cumey, cumez, val);
        frontbuffer(FALSE);
        popmatrix();
        break;
```

When you process the REDRAW token, you need to call `reshapeviewport()` in case the window has been resized.

■ C Program: REDRAW

```
/*  This program shows and manipulates the same cube in several
    different windows in double buffer mode.  Each window
    draws the cube slightly smaller than its predecessor.
    Thus, if there are three windows, the first window draws
    the cube full size (scale = 1/1), the second window
    draws the cube half size (scale = 1/2), and the third window
    draws the cube one-third size (scale = 1/3).  To run the
    program, type
        multicube  number
    where number is the number of windows you wish to open.
    You must be in the window manager to run this program.

    Interface:
        You must attach to one of the windows.
        Pressing the LEFT MOUSE button while moving
        the mouse from left to right (changing the X valuator)
        causes rotation in the x axis.
        Pressing the MIDDLE MOUSE button while moving
        the mouse from left to right causes rotation
        in the y axis.
        Pressing the RIGHT MOUSE button while moving
        the mouse from left to right causes rotation
        in the z axis.
        Pressing ESCKEY exits the program. */

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

```
/* scale factor associated with each window id */
static float   scalewin[50];

main (argc, argv)
int     argc;
char  *argv[];
{
    Icoord x;
    short  i, oldx, active;
    register   dx;
    short  val;
    Angle  cumex, cumey, cumez;  /*  rotation angles*/
    int     nwins; /*  how many windows for the cube */
    int     win_ids[10];/*  window ids--there is a maximum of
           10 windows per process.          */
    int     multiwindow;

    if (argc < 2) {
    printf ("Usage: multicube numberofwindows\n");
    exit  (0);
    }
    if (!ismex()) {
    printf ("You must be in the window manager to run multicube\n")
    exit(0);
    }

    cumex = 0; cumey = 0; cumez = 0;
    active = FALSE;

/*  count how many windows to open    */
    sscanf (argv[1], "%d", &nwins);

/*  limit number of windows to somewhere between 1 and 10*/
    if (nwins <= 0)  {
    printf ("Can't open fewer windows than one;\n");
    printf ("opening one window\n");
    nwins = 1;
```

```
    }
    else if (nwins > 10) {
    printf ("At most, ten windows can be opened per process\n");
    nwins = 10;
    }

/*  open the windows                          */
    for (i = 0; i < nwins; i++) {
    win_ids[i] = winopen ("cubes");
    if (i == 0) {
        doublebuffer();
        gconfig();
    }
    perspective (900, 1.0, 1.0, 3.5);
    translate (0.0, 0.0, -2.0);
    }

    qdevice (REDRAW);
    qdevice (INPUTCHANGE);
    qdevice (LEFTMOUSE);
    qdevice (MIDDLEMOUSE);
    qdevice (RIGHTMOUSE);
    qdevice (MOUSEX);
    qdevice (ESCKEY);

/*  for every window, calculate the scale factor, and draw
    the scene initially                                   */
    for (i = 0; i < nwins; i++) {
    scalewin[win_ids[i]] = 1.0/(float) (i + 1);
    qenter(REDRAW,win_ids[i]);
    }

    while (1) {

    while (qtest ()) {
        switch (qread (&val)) {
        case ESCKEY:
            gexit();
            exit(0);
            break;
        case REDRAW:
```

```
/*  only redraw windows that need to be redrawn          */
            winset ((long) val);
            reshapeviewport ();
            pushmatrix();
            frontbuffer(TRUE);
            cube (WHITE, cumex, cumey, cumez, val);
            frontbuffer(FALSE);
            popmatrix();
            break;
        case INPUTCHANGE:
            if (val)
            active = TRUE;
            else {
            active = FALSE;
                for (i = 0; i < nwins; i++) {
                winset (win_ids[i]);
                pushmatrix ();
                frontbuffer(TRUE);
                cube (WHITE, cumex, cumey, cumez, win_ids[i]);
                frontbuffer(FALSE);
                popmatrix ();
            }
            }
            break;
        case MOUSEX:
            oldx = x;
            x = val;
            dx = x - oldx;
            if (getbutton(LEFTMOUSE))      /* rotate x */
            cumex = cumex + dx;
            else if (getbutton(MIDDLEMOUSE)) /* rotate y */
            cumey = cumey + dx;
            else if (getbutton(RIGHTMOUSE))  /* rotate z */
            cumez = cumez + dx;
            break;
        default:
            break;
        }
/*  swapbuffers, even when program is inactive          */
        if (!active)
        while (!qtest()) swapbuffers();
```

```
    }
        for (i = 0; i < nwins; i++) {
        winset (win_ids[i]);
        pushmatrix ();
        cube (WHITE, cumex, cumey, cumez, win_ids[i]);
        popmatrix ();
    }
    swapbuffers();
    }  /*  while  */
}

/*  draw cube with color, rotation, and scale factor*/

cube (hue, cumex, cumey, cumez, winid)
Colorindex hue;
Angle cumex, cumey, cumez;
int winid;
{
    color (BLACK);
    clear ();
    rotate (cumex, 'x');
    rotate (cumey, 'y');
    rotate (cumez, 'z');
    scale (scalewin[winid], scalewin[winid], scalewin[winid]);
    color(hue);
    move (-1.0, -1.0, -1.0);
    draw (-1.0, -1.0, 1.0);
    draw (1.0, -1.0, 1.0);
    draw (1.0, 1.0, 1.0);
    draw (-1.0, 1.0, 1.0);
    draw (-1.0, -1.0, 1.0);
    move (-1.0, -1.0, -1.0);
    draw (-1.0, 1.0, -1.0);
    draw (1.0, 1.0, -1.0);
    draw (1.0, -1.0, -1.0);
    draw (-1.0, -1.0, -1.0);
    move (1.0, -1.0, -1.0);
    draw (1.0, -1.0, 1.0);
    move (-1.0, 1.0, -1.0);
    draw (-1.0, 1.0, 1.0);
```

```
    move (1.0, 1.0, -1.0);
    draw (1.0, 1.0, 1.0);
    move (1.0, 1.0, 0.0);
    draw (-1.0, 1.0, 0.0);
    draw (-1.0, -1.0, 0.0);
    draw (1.0, -1.0, 0.0);
    draw (1.0, 1.0, 0.0);
}
```

SAMPLE CODE

EXAMPLES

1: Getting Started S-1

2: Common Drawing Commands S-5

3: Object Coordinate Systems S-15

4: Double Buffer Display Mode S-19

5: Input/Output Devices S-23

6: Interactive Drawing S-29

7: Pop-up Menus S-33

8: Modeling Transformations S-47

9: Writemasks and Color Maps S-61

10: A Color Editor S-67

WORKSHOPS

1: diamond1.c S-75

2: color2.c S-79

3: double3.c S-83

4: overlay4.c S-87

5: poll5.c S-91

6: aim6.c S-95

7: queue7.c S-99

8: menu8.c S-105

9: threed9.c S-111

10: coord10.c S-117

11: translate11.c S-123

12: composite12.c S-129

13: view13.c S-135

14: parabola14.c S-141

Example 1: Getting Started

Getting your first graphics program to run on an IRIS is much like getting your first program to run on any other computer — the problems will be more involved with learning to edit a file, to compile and load a program with the proper libraries, and finally, to get it to run. Thus, the first program in this tutorial is not interesting at all from a graphics point of view, but it will probably be harder to get running than any of the other programs in the tutorial.

The first example draws a small red box in the lower left corner of the screen:

```
/* "first.c" */
#include "gl.h"

main()
{
    ginit();
    cursoff();
    color(BLACK);
    clear();
    color(RED);
    move2i(20, 20);
    draw2i(50, 20);
    draw2i(50, 50);
    draw2i(20, 50);
    draw2i(20, 20);
    gexit();
}
```

The very first line:

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

should be included in all graphics programs, as it defines type definitions, useful constants, and external definitions for all commands.

The graphical part of every program should begin with a call to `ginit()` (or `gbegin()`) and should end with `gexit()`. If you were to write a program to drive the raw hardware on the IRIS, you would find that there are literally hundreds of things you would have to initialize before you could draw the simplest thing on the screen. `ginit()` initializes the hardware and the software to a reasonable state so that a program as simple as the one above can run. (Note that `gbegin()` does everything `ginit()` does except it does not initialize the color map.)

This "reasonable" state includes definitions of the eight colors BLACK, WHITE, RED, GREEN, BLUE, YELLOW, CYAN, and MAGENTA. `ginit()` also sets up the screen as a 2-dimensional space with coordinates running from 0 to 1023 in the x-direction and 0 to 767 in the y-direction. Every pixel has size 1 in both the x- and y- directions. The origin is at the lower left corner of the screen.

The `cursoff()` command turns the cursor off (the default state has the cursor on). If you omit this command and try to draw across the cursor, you may get cursor glitches. After you get the program running, try deleting this command, and modifying the program so it draws a single line from (0, 0) to (1023, 767). The default cursor is in the center of the screen, so this line will draw across it.

All drawing is done in the current color, which is set by the `color()` command. "Drawing" means any command that draws something on the screen, including polygons, lines, text, points, pixels, and even the `clear` command. Thus the pair

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

clears the entire screen to black. `color(RED)` changes the current color to red. If this command were not in "first.c", the small box would be drawn in black and would not be visible.

The main part of "first.c" is made up of `move2i()` and `draw2i()` commands. The "2" stands for 2-dimensional, and the "i" means that the data are 32-bit integers (not floating-point numbers or 16-bit short integers). `move2i(x,y)` changes the current graphics position to the point (x, y); `draw2i(x,y)` draws a line from the current graphics position to the point (x, y), and changes the current graphics position to the new point. If you think of an imaginary pen doing the drawing, both `move2i(x,y)` and `draw2i(x,y)` put the pen at (x, y), but the `move` command causes it to be lifted before it is moved.

Note that many commands cause the current graphics position to become undefined. `clear()`, for example, does so. The sequence:

```
move2i(0, 500);
color(BLACK);
clear();
color(RED);
draw2i(500, 500);
```

will probably not draw a horizontal line — the `clear()` changes the current graphics position.

Example 2: Common Drawing Commands

The program "drawing.c" uses many of the more common drawing commands, including those to draw filled and unfilled polygons, circles, arcs, points, rectangles, and text. Note that after `ginit` is called, the textport is set to a small area in the lower left corner of the screen. This prevents the textport from covering up most of the screen when the program exits. To return the textport to its original size and location, the following program can be executed:

```
/* "tpbig.c" */
#include "gl.h"

main()
/* initialize the textport to be 40 lines and 80 columns */
{
    ginit();
    textinit();
    gexit();
}
```

This is the FORTRAN version of "tpbig":

```
C
C     initialize the textport to be 40 lines and 80 columns
C
C
      CALL GINIT()
      CALL TEXTIN()
      CALL GEXIT()
C
```

```
      STOP
      END
```

This is the code for "drawing.c":

```
/* "drawing.c" */
#include "gl.h"

long cone[][2] = {100, 300,
                  150, 100,
                  200, 300};

char *singlechar = "X";

main()
{
    register long i, j;

    ginit();
    textport(50,300,50,200);
        /* the textport is set to a small area in the lower
         * left corner of the screen so that when the program is
         * finished, the textport will not cover up the image */
    cursoff();

    /* draw an ice-cream cone */

    color(WHITE);
    clear();
    color(YELLOW);
    polf2i(3, cone);     /* draw the ice-cream cone */
    color(GREEN);        /* first scoop is mint */
    arcfi(150, 300, 50, 0, 1800);/* only half of it shows */
    color(RED);          /* second scoop is cherry */
    circf(150.0, 400.0, 50.0);
    color(BLACK);
    poly2i(3, cone);     /* outline the cone in black */

    /* Next, draw a few filled and unfilled arcs in the upper
```

```
 * left corner of the screen.
 */

arcf(100.0, 650.0, 40.0, 450, 2700);
arci(100, 500, 40, 450, 2700);

arcfi(250, 650, 80, 2700, 450);
arc(250.0, 500.0, 80.0, 2700, 450);

/* Now, put up a series of filled and unfilled rectangles
 * with the names of their colors printed inside of them
 * across the rest of the top of the screen.
 */

color(GREEN);
recti(400, 600, 550, 700);
cmov2i(420, 640);
charstr("Green");

color(RED);
rectfi(600, 600, 800, 650);
color(BLACK);
cmov2(690.0, 620.0);
charstr("Red");

color(BLUE);
rect(810.0, 700.0, 1000.0, 20.0);
cmov2i(900, 300);
charstr("Blue");

/* Now draw some text with a ruler on top
 * to measure it by. */

/* First the ruler: */

color(BLACK);

move2i(300, 400);
draw2i(650, 400);
for (i = 300; i <= 650; i += 10) {
    move2i(i, 400);
```

```
    draw2i(i, 410);
}

/* Then some text: */

cmov2i(300, 380);
charstr("The first line is drawn ");
charstr("in two parts.");

cmov2i(300, 368);
charstr("This line is only 12 pixels lower.");

cmov2i(300, 354);
charstr("Now move down 14 pixels ...");

cmov2i(300, 338);
charstr("And now down 16 ...");

cmov2i(300, 320);
charstr("Now 18 ...");

cmov2i(300, 300);
charstr("And finally, 20 pixels.");

/* Finally, show off the entire font.  The cmov2i() before
 * each character is necessary in case that character
 * is not defined.
 */

for (i = 0; i < 4; i++)
    for (j = 0; j < 32; j++) {
        cmov2i(300 + 9*j, 200 - 18*i);
        *singlechar = (char)(32*i + j);
        charstr(singlechar);
    }

for (i = 0; i < 4; i++) {
    cmov2i(300, 100 - 18*i);
    for (j = 0; j < 32; j++) {
        *singlechar = (char)(32*i + j);
        charstr(singlechar);
```

```
        }
    }

    gexit();
}
```

This is the FORTRAN version of "drawing":

```
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
C        Installation note:
C      Various Fortran compilers may require different
C      styles of INCLUDEs.
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      INCLUDE 'fgl.h'
C
      INTEGER CONE(2,3), I, J, STRLEN
      CHARACTER*1 ONECH, STRING*50
      DATA CONE /100,300,  150,100,  200,300/
C
      CALL GINIT
C        set the textport to a small area in the lower right
C        corner of the screen so that when the program is
C        finished, the textport will not cover
C        up the image
      CALL TEXTPO(650,900,50,200)
      CALL CURSOF
C
C        draw an ice cream cone
      CALL COLOR(WHITE)
      CALL CLEAR
C        draw the cone
      CALL COLOR(YELLOW)
      CALL POLF2I(3,CONE)
C        the first scoop is mint
      CALL COLOR(GREEN)
C        only half of it shows
      CALL ARCFI(150,300,50,0,1800)
C        the second scoop is cherry
      CALL COLOR(RED)
```

```
      CALL CIRCF(150.0,400.0,50.0)
      CALL COLOR(BLACK)
C       outline the cone in black
      CALL POLY2I(3,CONE)
C
C       next draw a few filled and unfilled arcs in the upper
C       left corner of the screen
      CALL ARCF(100.0,650.0,40.0,450,2700)
      CALL ARCI(100,500,40,450,2700)
      CALL ARCFI(250,650,80,2700,450)
      CALL ARC(250.0,500.0,80.0,2700,450)
C
C       Now, put up a series of filled and unfilled rectangles
C       with the names of their colors printed inside of them
C       across the rest of the top of the screen.
      CALL COLOR(GREEN)
      CALL RECTI(400,600,550,700)
      CALL CMOV2I(420,640)
      CALL CHARST('Green',5)
C
      CALL COLOR(RED)
      CALL RECTFI(600,600,800,650)
      CALL COLOR(BLACK)
      CALL CMOV2(690.0,620.0)
      CALL CHARST('Red',3)
C
      CALL COLOR(BLUE)
      CALL RECT(810.0,700.0,1000.0,20.0)
      CALL CMOV2I(900,300)
      CALL CHARST('Blue',4)
C
C       Now draw some text with a ruler on top to
C       measure it by.
C
C       First the ruler:
      CALL COLOR(BLACK)
      CALL MOVE2I(300,400)
      CALL DRAW2I(650,400)
      DO 100 I=300,650,10
        CALL MOVE2I(I,400)
        CALL DRAW2I(I,410)
```

```
100   CONTINUE
C
C       Then some text:
      CALL CMOV2I(300,380)
      STRING = 'The first line is drawn incorrectly '
      CALL CHARST(STRING,LEN(STRING))
      CALL CHARST('in two parts.',13)
C       NOTE:  Fortran pads STRING with spaces to its defined
C       length, and LEN(STRING) returns the defined length (50)
C       instead of thelength of the character substring
C       inserted into it.
C
      CALL CMOV2I(300,364)
      STRING = 'This line is drawn correctly '
      STRLEN = LEN('This line is drawn correctly ')
      CALL CHARST(STRING,STRLEN)
C       This is the only way (other than counting by hand, as
C       was done for the second part of this line, below) of
C       gettingthe length of the actual set of characters
C       to be printed.
      CALL CHARST('in two parts.',13)
C
      CALL CMOV2I(300,352)
      STRLEN = LEN('This line is only 12 pixels lower.')
      CALL CHARST('This line is only 12 pixels lower.',STRLEN)
C
      CALL CMOV2I(300,338)
      STRLEN = LEN('Now move down 14 pixels ...')
      CALL CHARST('Now move down 14 pixels ...',STRLEN)
C
      CALL CMOV2I(300,322)
      STRLEN = LEN('And now down 16 ...')
      CALL CHARST('And now down 16 ...',STRLEN)
C
      CALL CMOV2I(300,304)
      STRLEN = LEN('Now 18 ...')
      CALL CHARST('Now 18 ...',STRLEN)
C
      CALL CMOV2I(300,284)
      STRLEN = LEN('And finally, 20 pixels.')
      CALL CHARST('And finally, 20 pixels.',STRLEN)
```

```
C
C       Finally, show off the entire font.  The cmov2i() before
C       each character is necessary in case that character is
C       not defined.
      DO 300 I=0,3
        DO 200 J=0,31
          CALL CMOV2I(300 + 9*J, 200 - 18*I)
          ONECH(1:1) = CHAR(32*I + J)
          CALL CHARST(ONECH,1)
200     CONTINUE
300   CONTINUE
C
      DO 500 I=0,3
        CALL CMOV2I(300, 100 - 18*I)
        DO 400 J=0,31
          ONECH(1:1) = CHAR(32*I + J)
          CALL CHARST(ONECH,1)
400     CONTINUE
500   CONTINUE
C
900   CONTINUE
      IF(.NOT. GETBUT(RIGHTM))GO TO 999
      call color(BLACK)
      call clear
      call texti
      CALL GEXIT()
      STOP
      END
```

The first thing the program draws is an ice-cream cone in the left part of the screen. The cone is drawn with the polygon command `polf2i()`. All of the polygon commands begin with the prefix "pol". In this case, we want the polygon to be filled in with the current color, and the "f" stands for filled. The characters "2i" mean the same thing that they do for the move and draw commands: "2" means 2-dimensional, and "i" means 32-bit integer. An unfilled (wire-frame) polygon has a "y" instead of an "f".

The first argument to all the polygon commands is the number of vertices, and the second is an array of points. A polygon can have many vertices (the exact number varies, but if you use fewer than 500 you will be safe). Filled polygons must be convex, or you will get unpredictable results.

The first scoop on the ice cream cone is a half-circle. A half circle is a special case of an arc so we use the `arcfi()` command. As was the case with polygons, the "f" stands for filled and the "i" for 32-bit integer. The first two arguments to the arc commands are the x and y coordinates of the center of the arc; the third is the radius. The last two arguments are the starting and ending angles. In the IRIS graphics library, all angles are integers measured in of degrees, so an angle of 1800 is 180 degrees. Angles in the arc commands are measured counter-clockwise from the positive x-axis. More examples of arcs appear later in this sample program.

Circles come in the same variations as arcs — filled or unfilled, and floating-point, 32-bit integer, or 16-bit short integer. The three arguments to the circle commands are the same as the first three for the arc commands — x-center, y-center, and radius. In the sample program, we have used the floating-point version of the filled circle command for the sake of variety only — the integer versions would work fine.

Although it makes no difference yet, note that circles and arcs are 2-dimensional. They lie in the x-y plane with a z coordinate of zero. When we get to 3-dimensional examples, we will see that circles rotated out of the x-y plane appear to be ellipses.

The second part of the program draws two filled arcs and two unfilled arcs. Notice that filled arcs are filled from the center — like pieces of pie — and that unfilled arcs are simply a portion of a circle. The order in which the angles are specified is important; the arc is filled in counterclockwise from the starting angle to the ending angle.

Like circles and arcs, rectangles are 2-dimensional, with their z-coordinates equal to zero. The edges are parallel to the edges of the screen, so you need specify only the two opposite corners. Any pair of opposite corners will do, as the example illustrates. Rectangles come filled or unfilled, and in floating-point, integer, or short versions.

Character strings are drawn beginning at the current character position (which is different from the current graphics position). The commands `cmov()`, `cmovi()`, `cmovs()`, `cmov2()`, `cmov2i()`, and `cmov2s()` set the current character position. In the example program, we only use the 2-

dimensional versions of these commands. Notice that the second rectangle is filled with red. We have to change the color to something other than red so that the character string will show up on the red background.

The final examples in this program show how the current character position changes as characters are drawn. After each character string is printed, the current character position is automatically moved to follow the last character printed. The default font is 9 pixels wide and 16 high. The ruler above the text has a short cross mark every 10 pixels.

The next four lines of text show the default font. Note that the input to `charstr()` is a string, which in C is a null-terminated array of characters. The example uses a string with a single character in it, and replaces that character by the next character in the font. Odds are there is an explicit `cmov2i()` before each character is drawn because if the character is not defined in the font, nothing is drawn, and the current character position is not changed.

The whole font is drawn again below the first copy without the `cmov2i()`. Only characters 1 and 7 in the range 0-31 of the ASCII character set Interrupt ASCII 1 (control-A) prints a solid rectangle, and ASCII 7 (control-G, or BELL) prints a little picture of a bell.

Example 3: Object Coordinate Systems

For many applications (in fact, probably most), screen coordinates are not the most convenient to use. It is usually more convenient to describe graphical objects in terms of their own coordinate systems, and then map those coordinate systems to the screen. The example program "doily.c" illustrates how the `ortho2()` command is used to perform such a mapping. In addition, it shows how the `viewport` command specifies the area of the screen where drawing takes place. Note that the textport is set to a small area in the lower left corner of the screen as it was in the previous example. It can be reset to its original size and location by calling the "tpbig.c" program.

```
/* "doily.c" */
#include "gl.h"
#include <math.h>

#define PI 3.1415926535

main(argc, argv)
int argc;
char *argv[];
{
    long numpts, i, j;
    float points[100][2];

    /* First figure out how many points there are. */

    if (argc != 2) {
        printf("Usage: %s <point count>\n", argv[0]);
        exit(0);
```

```
    }

    numpts = atoi(argv[1]);

   /* convert argument to internal format */

    if (numpts > 100) {
        printf("Too many points\n");
        exit(0);
    }

    if (numpts < 3) {
        printf("Too few points\n");
        exit(0);
    }
    /* Now get the x and y coordinates of numpts equally-
     * spaced points around the unit circle.
     */

    for (i = 0; i < numpts; i++) {
        points[i][0] = cos((i*2.0*PI)/numpts);
        points[i][1] = sin((i*2.0*PI)/numpts);
    }

    ginit();
    textport(30,200,30,200);
    cursoff();

    color(WHITE);
    clear();                /* clear the whole screen */

    viewport(200, 800, 100, 700);    /* restrict to a
                                        square viewport */
    color(BLACK);
    clear();
    color(RED);

    ortho2(-1.2, 1.2, -1.2, 1.2);
```

```
    for (i = 0; i < numpts; i++)
        for (j = i+1; j < numpts; j++) {
            move2(points[i][0], points[i][1]);
            draw2(points[j][0], points[j][1]);
        }

    gexit();
}
```

Typing "doily 21" will print a 21-point doily. A doily connects all pairs of evenly spaced points around a circle with straight lines. The first part of the program is standard UNIX to figure out how many vertices to use. We have arbitrarily restricted the number of vertices to be between 3 and 100. The next part of the program calculates the x and y coordinates of a set of `numpnts` equally spaced points on the unit circle. (The unit circle has radius 1.0 and is centered at the origin: x = 0.0, y = 0.0.)

The entire screen is cleared to white, and a `viewport()` command constrains drawing to a square region in the center of the screen (200 <= x <= 800, and 100 <= y <= 700). Notice that the next clear will satisfy that constraint, and a square black region will be filled.

The `ortho2()` command sets things up so that the region (-1.2 <= x, y <= 1.2) completely fills the viewport. The unit circle will therefore fit into the viewport. Finally, red lines are drawn in red between all possible pairs of points.

The `viewport()` command specifies the portion of the physical screen to be used for output, and the `ortho2()` command describes the world (or object) coordinate range that will fill that part of the screen. One command tells the part of the world to view, and the second tells where on the screen to draw it. There is a "2" in `ortho2()` because it is a 2-dimensional version. There is an `ortho()` command that will be described later.

Usually, the aspect ratios (the ratio of the x length to the y length) of the `viewport()` and the `ortho2()` commands will be the same (here, they are both square), but this is not required. If they are different, the drawing will be squashed in the x or the y direction. To see clipping in action, try changing the `ortho2()` command to `ortho2(-0.9,0.9,-0.9,0.9)`.

Example 4: Double Buffer Display Mode

The "bounce.c" program is the first dynamic example in this tutorial. A ball (a circle) bounces around on a rectangular piece of the screen. The program is simple, and introduces one new feature — double buffering.

In double buffer mode, the available bitplanes on your system are divided into two equal sets, and only one of the sets is viewed. In an eight-bitplane system in double buffer mode, for example, four bitplanes are assigned to the front buffer and four to the back buffer. Only the contents of the front buffer are displayed, and all writing is normally done into the back buffer.

To get the IRIS into double buffer mode, two commands are required: `doublebuffer()` and `gconfig()`. `doublebuffer()` tells the IRIS something about the display mode to use, and `gconfig()` changes to the new mode. The reason that two commands are required is that the display mode can be changed in other ways, and `gconfig()` is only called after all of the changes have been specified. Some sets of modes are inconsistent, and you must be able to specify the new setup completely before changing to it. For now, just remember to follow the `doublebuffer()` command by a `gconfig()` command, or you will remain in single buffer mode.

Before starting, we would like to clear the screen to white. Since there are two buffers, the obvious way to do this is to clear one buffer, swap buffers, and then clear the other buffer. Clearing large portions of the screen is one of the slowest things the IRIS does, so in general, this should be avoided. In this case, the two clears would occur only once at the beginning of the program, so it would not hurt too much.

As was stated earlier, there is a front buffer and a back buffer, and normally the contents of the front buffer are displayed while the contents of the back buffer are being updated. `frontbuffer(TRUE)` tells the IRIS to write

into the front buffer as well as the back buffer. `frontbuffer(FALSE)` turns off writing into the front buffer. The sequence:

```
frontbuffer(TRUE);
clear();
frontbuffer(FALSE);
```

has the net effect of clearing both the front and back buffers.

There is a `backbuffer()` command with similar effects, but it is not used as much. `ginit()` effectively does a `frontbuffer(FALSE)` and a `backbuffer(TRUE)` command.

The next two lines in the sample program illustrate the cure for another common problem. You would like to set a viewport that is not the entire screen, but you would also like to have the screen coordinates correspond to the world coordinates. The problem is solved by adding 0.5 to the x and y maximums in the `ortho2` command, and subtracting 0.5 from the x and y minimums. To see why this is necessary, consider just the x coordinates. A `viewport(100,200,100,200)` command says to use the screen coordinates from 100 to 200, inclusive. This is 101 pixels! If you use the command `ortho2(100.0,200.0,100.0,200.0)`, the world x range is 100.0, so changing the world coordinates by 100 will change the screen range by 101. This may cause round-off problems, especially if you are trying to deal with individual pixels. If, instead, you use `ortho2(99.5,200.5,99.5,200.5)`, things will match up perfectly. `ginit()` originally issues the command `ortho2(-0.5,1023.5,-0.5,767.5)`. You might think that you could just add 1 to the maximum values and have things work out, but this will make every integer fall exactly on the boundary between two screen pixels. Rounding would occur for every pixel with unpredictable results.

Once the "bounce.c" program is initialized, it goes into an infinite loop and repeatedly recalculates the position of the ball, redraws it in the back buffer, and swaps the back and front buffers. The calculation in this simple program is straight-forward — increment the position by the velocity, and if the ball has rammed into a wall, flip the velocity to the other direction.

In most dynamic graphics, one first clears the screen, and then draws in the next view. If the view is complicated, the drawing process can block because the `clear()` command takes so long. If we do the `clear()`

before the calculations, the hardware can start the clearing and get some of it done while the next frame is being calculated. After the next frame is drawn, a `swapbuffers()` command exchanges the front and back buffers.

The sample program runs forever, and must be terminated by interrupting the system. In the next example, we will provide a cleaner exit mechanism.

"bounce.c" follows:

```
/* "bounce.c" */
#include "gl.h"

#define XMIN 100          /* XMIN,...define the region of the */
#define YMIN 100          /* screen where the ball bounces. */
#define XMAX 900
#define YMAX 700

main(argc, argv)
int argc;
char *argv[];
{
    long xpos = 500, ypos = 500;
    long xvelocity, yvelocity, radius;

    if (argc != 4) {
        printf("Usage: %s <xvelocity> <yvelocity>
                <radius>\n", argv[0]);
        exit(0);
    }

    xvelocity = atoi(argv[1]);
    /* convert the ascii values of the */
    yvelocity = atoi(argv[2]);
    /* parameters to internal integer */
    radius = atoi(argv[3]);     /* format */
    if (radius <= 0)    /* sanity check */
        radius = 10;

    ginit();
    doublebuffer();
    gconfig();
    color(WHITE);
```

```
    frontbuffer(TRUE);
    clear();
    frontbuffer(FALSE);
    viewport(XMIN, XMAX, YMIN, YMAX);
    ortho2(XMIN - 0.5, XMAX + 0.5, YMIN - 0.5, YMAX + 0.5);
    while(1) {
        color(BLACK);
        clear();
        xpos += xvelocity;
        ypos += yvelocity;
        if (xpos > XMAX - radius ||
            xpos < XMIN + radius) {
            xpos -= xvelocity;
            xvelocity = -xvelocity;
        }
        if (ypos > YMAX - radius ||
            ypos < YMIN + radius) {
            ypos -= yvelocity;
            yvelocity = -yvelocity;
        }
        color(YELLOW);
        circfi(xpos, ypos, radius);
        swapbuffers();
    }
}
```

Example 5: Input/Output Devices

This example shows how to make use of the mouse to control a dynamic graphics program. We will modify the previous example, "bounce.c", as follows:

- The program will start with the ball at rest.
- When the left mouse button is pressed, the x velocity will increase by 1.
- The middle mouse button increases the y velocity by 1.
- The right mouse button stops the ball.
- Pressing the left and right mouse buttons simultaneously will exit the program.

An easy way to make this user interface work is to say an event occurs every time that all of the mouse buttons are up. Between times when they are all up, some subset of the mouse buttons will have gone down. The set of all mouse buttons that have been down between events will be the result of the event.

The routine `checkmouse()` is called every time the picture is drawn. `checkmouse()` looks to see if the state of any mouse button has changed, and records the results. If an event occurs as a result of such a change, velocities are adjusted accordingly.

In `checkmouse()`, the variable `buttons` records the current set of buttons that are down, and `pressed` records the total number of buttons pressed since the last event. We queue the mouse buttons so that no button events are missed. In both `buttons` and `pressed`, we use the 1, 2, and 4 bits to record the state of the `LEFT`, `MIDDLE`, and `RIGHT` mouse buttons, respectively. If we logically OR `LEFT` with `pressed` the `LEFT` bit is turned on in that variable. Similarly, logically XORing `LEFT` with

buttons changes the state of the LEFT bit in the buttons variable. When buttons finally gets to zero (all buttons up), the set of all buttons pressed between events is recorded in pressed.

With a user interface like this, it is often a good idea to make the combination of all three buttons be an abort command. This way, if you accidentally press down the wrong button combination and notice it before you release them, you can just push down the rest of the buttons, and abort the entire command.

In the main routine, the three calls to qbutton() instruct the IRIS to record all changes of state of the mouse buttons (either down or up) in the event queue.

Another interesting thing to note is the use of the qtest() command. If we were to call qread() immediately, then everything would stop until a mouse button is pressed or released — the motion would stop. qtest() will return zero if nothing has happened, so the routine checkmouse() will just return.

The program as written can still be improved. It assumes that when things start up, all of the mouse buttons are up. It will behave in a very strange manner if you hold down a mouse button or two, and start the program. The sense of those buttons will be reversed. To fix this, you can add this line just before the three qdevice() commands:

```
whilegetbutton(LEFTMOUSE)|getbutton(MIDDLEMOUSE)|getbutton(RIGHTMOUSE));
```

This will cause the program to spin its wheels until all mouse buttons are released.

Here is a listing of the "iobounce.c" program:

```
/* "iobounce.c" */
#include "gl.h"
#include "device.h"

#define XMIN 100
#define YMIN 100
#define XMAX 900
#define YMAX 700
```

```
long xvelocity, yvelocity;

main(argc, argv)
int argc;
char *argv[];
{
    long xpos = 500, ypos = 500;
    long radius;

    xvelocity = yvelocity = 0;
    radius = 10;

    ginit();
    doublebuffer();
    gconfig();
    color(WHITE);
    frontbuffer(TRUE);
    clear();
    frontbuffer(FALSE);
    viewport(XMIN, XMAX, YMIN, YMAX);
    ortho2(XMIN - 0.5, XMAX + 0.5, YMIN - 0.5, YMAX + 0.5);

    qdevice(LEFTMOUSE);
    qdevice(MIDDLEMOUSE);
    qdevice(RIGHTMOUSE);

    while(1) {
        color(BLACK);
        clear();
        checkmouse();
        xpos += xvelocity;
        ypos += yvelocity;
        if (xpos > XMAX - radius ||
            xpos < XMIN + radius) {
            xpos -= xvelocity;
            xvelocity = -xvelocity;
        }
        if (ypos > YMAX - radius ||
            ypos < YMIN + radius) {
            ypos -= yvelocity;
            yvelocity = -yvelocity;
```

```
        }
        color(YELLOW);
        circfi(xpos, ypos, radius);
        swapbuffers();
    }
}

#define LEFT 1
#define MIDDLE 2
#define RIGHT 4

checkmouse()
{
    static buttons = 0, pressed = 0;
    Device val;

    while (qtest()) {
        switch (qread(&val)) {
            case LEFTMOUSE:
                buttons ^= LEFT;
                pressed |= LEFT;
                break;
            case MIDDLEMOUSE:
                buttons ^= MIDDLE;
                pressed |= MIDDLE;
                break;
            case RIGHTMOUSE:
                buttons ^= RIGHT;
                pressed |= RIGHT;
                break;
        }
        if (buttons == 0) {
            switch (pressed) {
                case LEFT:/* increase xvelocity */
                    if (xvelocity >= 0)
                        xvelocity++;
                    else
                        xvelocity--;
                    break;
                case MIDDLE:/* increase yvelocity */
                    if (yvelocity >= 0)
```

```
                        yvelocity++;
                    else
                        yvelocity--;
                    break;
                case RIGHT:      /* stop ball */
                    xvelocity = yvelocity = 0;
                    break;
                case LEFT+RIGHT:/* exit */
                    gexit();
                    exit(0);
            }
        pressed = 0;
        }
    }
}
```

Example 6: Interactive Drawing

This example is a simple interactive drawing program. Beginning with the screen cleared to black, you can move the cursor around with a mouse, and mark endpoints of lines to be drawn. The middle button sets the current graphics position to the cursor position; the left button draws from the current graphics position to the cursor position and then sets the current graphics position to the cursor position. A connected set of segments can be drawn by repeatedly moving the mouse and pressing the left button. The program exits when the right button is pressed.

Here is the source code:

```
/* "draw.c" */
#include "gl.h"
#include "device.h"

main()
{
    Device val, xpos, ypos;

    ginit();
    color(BLACK);
    cursoff();
    clear();
    curson();
    color(RED);
    qdevice(LEFTMOUSE);
    qdevice(MIDDLEMOUSE);
    qdevice(RIGHTMOUSE);
    tie(LEFTMOUSE, MOUSEX, MOUSEY);
    tie(MIDDLEMOUSE, MOUSEX, MOUSEY);
```

```
    while(1) {
      switch(qread(&val)) {/* wait for mouse down */
         case RIGHTMOUSE:       /* quit */
           gexit();
           exit(0);
         case MIDDLEMOUSE:      /* move */
           qread(&xpos);
           qread(&ypos);
           move2i(xpos, ypos);
           qread(&val);/* these three reads clear out */
           qread(&val);/* the button up report */
           qread(&val);
           break;
         case LEFTMOUSE:        /* draw */
           qread(&xpos);
           qread(&ypos);
           cursoff();
           draw2i(xpos, ypos);
           curson();
           qread(&val);
           qread(&val);
           qread(&val);
           break;
        }
    }
}
```

The program is controlled by the mouse. The right button exits, the middle button does a move to the current cursor position, and the left button draws to the current cursor. All input/output is done using the event queue.

All of the mouse buttons are queued, and the left and middle buttons are tied to the x- and y- coordinates of the mouse. The `tie()` command guarantees that when the left or middle mouse buttons change (go up or down), the values of `MOUSEX` and `MOUSEY` will be recorded in the event queue. The two entries in the queue after the button event will be the `MOUSEX` and `MOUSEY` values in that order. Thus the code in the case statement does not have to check to see that the following events are of type `MOUSEX` and `MOUSEY`.

The main part of the program is a loop that reads the event queue with `qread()`, and quits, moves, or draws, depending on the event recorded there. In the case of the `MIDDLEMOUSE` event, the xposition and yposition are read, and a move to those coordinates is issued. The next three `qread()` commands throw away the event where the mouse button comes up. Remember that when the button comes up, there will be a button event and then the `MOUSEX` and `MOUSEY` events.

The draw code is similar, except that the cursor must be turned off and on before each draw. This is because of the way the cursor works. The hardware draws the cursor directly onto the bitplanes, clobbering whatever was underneath it. To restore the image after drawing the cursor, the screen area where the cursor is to be drawn is first copied away. When the cursor moves to a new spot, the copy is written back onto the screen (thus erasing the cursor), the space under the new cursor position is copied away, and the cursor is drawn again. The problem is that if the cursor has been drawn, and something is drawn over it, when the cursor is erased by copying back the old data, the new drawing is also lost. In this program, the problem would be severe, since all of the drawing will be in the vicinity of the cursor.

Turning the cursor off erases the cursor by replacing the area it covered by the good copy. Drawing is done with no cursor, and when the cursor is turned back on, an updated version of the screen area is saved. Try commenting out the `cursoff()` and `curson()` commands if you like to see what happens.

As it stands, the program could be improved. It assumes that once a mouse button is pressed, it will be released before another button is pressed. To work correctly in all cases, the three `qread()` commands that throw away the upstroke should be replaced by code that makes sure that it was the same button that went down.

Example 7: Pop–up Menus

The program "popup.c" demonstrates how a simple pop–up menu can be implemented. The heart of the program is the routine `popup()` that returns the number of the user's selection. The main program itself is extremely simple — after some initialization, it goes into a loop where it waits for a pop–up menu event, and dispatches to the appropriate routine. Depending on which pop–up entry the user selects, it draws a line, some points, a circle, a filled or unfilled rectangle, or exits.

The pop–up menu is drawn when the left mouse button is pressed down. While the button is held down, the menu remains on the screen, and as the cursor moves across a menu entry, that entry is highlighted. When the button is released, the entry under the cursor at that time is selected. If the cursor is outside the menu when the button is released, `popup()` returns the value 0. There are many other ways a pop–up menu could be implemented, and the routine `popup()` can easily be modified to change its behavior.

The first thing to notice about the program is the way that the pop–up menus make use of the color map. Here, we make use of two bitplanes for the pop–up menu. In this simple example, the main program itself uses only one bitplane — it only draws in black and white, and assuming that no pop–up menus were used, it could get away with two `mapcolor()` commands:

```
mapcolor(0, 0, 0, 0);
mapcolor(1, 255, 255, 255);
```

The color map is set up for the pop–up menus as follows: Pixels on the screen that are not under the pop–up menu have the bits in bitplanes 1 and 2 equal to zero (Bitplane 0 is used for the drawing). When either of the bits in bitplanes 1 or 2 is one, bit zero has no effect. This is done by mapping both

combinations to the same thing. There are three combinations with at least one bitplane non-zero: 10, 01, and 11. We use these three combinations to represent menu background, menu text, and highlighted menu. The map is set up as follows:

```
BP2 BP1 BP0
--- --- ---
 0   0   0  No menu, no drawing--black (r = 0, g = 0, b = 0)
 0   0   1  No menu, drawing--green (r = 0, g = 255, b = 0)
 0   1   0  Menu background, no drawing--black (r = 0, g = 0, b = 0)
 0   1   1  Menu background, drawing--black (r = 0, g = 0, b = 0)
 1   0   0  Menu text, no drawing--white (r = 255, g = 255, b = 255)
 1   0   1  Menu text, drawing--white (r = 255, g = 255, b = 255)
 1   1   0  Menu highlight, no drawing--grey (r = 100, g = 100, b = 100)
 1   1   1  Menu highlight, drawing--grey (r = 100, g = 100, b = 100)
```

The first eight `mapcolor()` commands in the main routine set up this color map. If the main drawing uses more than two colors (say eight), then there will be eight mapcolors for each of the three menu combinations to make the menu show up, regardless of the contents of the other planes.

The cursor draws in all three planes (writemask = 7) with color 4 (which is mapped to white) so the white cursor will always show up, no matter what combination of drawing and menu appears under it. Since the cursor is drawn in the same planes as everything else, it must be turned on and off every time a drawing operation is performed.

The sample program uses only the left mouse button, and will need to know where the mouse was when the button was pressed, so the button itself is queued, and tied to the x- and y- mouse coordinates.

Next, the main program clears the screen to black, and resets the viewport to a square near the middle of the screen, and uses an `ortho` command that maps world coordinates between -1.0 and 1.0 to that viewport.

Finally, the program goes into a loop, waiting for a menu event, and drawing the appropriate thing. All of the other interesting code is in the `popup()` routine.

`popup()` takes a menu structure as a parameter that includes the text of the menu, draws the menu under the cursor, waits for the selection to take place, and then returns the value of the selection. The menu structure is an array of pairs of entries, the first of which is an identifying number, and the second a text string. When a menu entry is picked, the identifying number is returned. It would have been easy to return the position of the entry in the menu, but attaching identifying numbers often makes things much easier to program.

The `popup()` routine needs to do a lot of things. When it is invoked, it has no idea what the current writemask, color, viewport, and matrix are, and it has to change all of these. Therefore, it saves away all that information, and restores it upon exit.

Next, it has to count the entries in the menu and calculate the size of the menu, and where on the screen to place the menu. It will try to center the menu under the cursor, but if the cursor is pressed near an edge or corner of the screen, entire menu will appear on the screen.

The menu is then drawn, restricted to the two menu bitplanes. `popup()` then goes into a loop waiting for the mouse button to come up. Before checking the event queue each time, it reads the current x- and y- locations of the cursor and checks to see if a highlighting change is necessary. The variable `lasthighlight` contains the number of the menu entry last highlighted — (0 <= `lasthighlight` < `menucount` — and is -1 if none of the entries are highlighted. If the cursor has moved to cover a new menu entry, the last highlighted entry (if there was one) is changed back to the normal color, and the new entry is highlighted. Note that the rectangle clears for erasing and redrawing are carefully chosen not to obliterate the boundary lines.

Finally, if the mouse button was released, the routine reads the current position, determines what menu entry it is above, if any, and returns the corresponding identifying number or zero.

For a general-purpose menu package, there really should be checks to see if the queue event is the correct mouse button, because a general menu package will have no idea what other events the user has queued.

The C version of "popup" follows:

```
/* "popup.c" */
#include "gl.h"
```

```
#include "device.h"
#define LINE 1
#define POINTS 2
#define CIRCLE 3
#define RECT 4
#define RECTF 5
#define QUIT 6
typedef struct {
    short type;
    char *text;
} popupentry;

popupentry mainmenu[] = {
    {LINE, "Line"},
    {POINTS, "100 points"},
    {CIRCLE, "Filled circle"},
    {RECT, "Outlined rectangle"},
    {RECTF, "Filled rectangle"},
    {QUIT, "Quit"},
    {0, 0}        /* mark end of menu */
};
main()
{
    register i, j;
    short command;

    ginit();
    mapcolor(0, 0, 0, 0);          /*background only*/
    mapcolor(1, 0, 255, 0);        /*drawing only*/
    mapcolor(2, 0, 0, 0);          /*popup background*/
    mapcolor(3, 0, 0, 0);          /*popup background over drawing
    mapcolor(4, 255, 255, 255); /*popup text only*/
    mapcolor(5, 255, 255, 255); /*popup text over drawing*/
    mapcolor(6, 100, 100, 100); /*popup highlight only*/
    mapcolor(7, 100, 100, 100); /*popup highlight over drawing*
    setcursor(0, 4, 7);
    qdevice(LEFTMOUSE);
    tie(LEFTMOUSE, MOUSEX, MOUSEY);

    cursoff();
    color(0);
```

```
    clear();
    curson();
    viewport(150, 850, 50, 750);
    ortho2(-1.0, 1.0, -1.0, 1.0);
    while (1) {
        command = popup(mainmenu);
        cursoff();
        color(0);
        clear();
        color(1);
        switch(command) {
            case LINE:
                move2(-1.0, -1.0);
                draw2(1.0, 1.0);
                break;
            case POINTS:
                for (i =  0; i < 10; i++)
                    for (j = 0; j < 10; j++)
                       pnt2(i/20.0, j/20.0);
                break;
            case CIRCLE:
                circf(0.0, 0.0, 0.5);
                break;
            case RECT:
                rect(-0.5, -0.5, 0.5, 0.5);
                break;
            case RECTF:
                rectf(-0.5, -0.5, 0.5, 0.5);
                        break;
            case QUIT:
                greset();
                        gexit();
                exit(0);
        }
        curson();
    }
}

popup(names)
popupentry names[];
{
```

```
register short i, menucount;
short menutop, menubottom, menuleft, menuright;
short lasthighlight = -1, highlight;
Device dummy, x, y;
short savecolor, savemask;
short llx, lly, urx, ury;

menucount = 0;
qread(&dummy);
qread(&x);
qread(&y);
savecolor = getcolor();/* save the state of everything */
savemask = getwritemask();
getviewport(&llx, &urx, &lly, &ury);
pushmatrix();
viewport(0, 1023, 0, 767); /* now setup to draw the menu */
ortho2(-0.5, 1023.5, -0.5, 767.5);
while (names[menucount].type)
    menucount++;
menutop = y + menucount*8;
menubottom = y - menucount*8;
if (menutop > 767) {
    menutop = 767;
    menubottom = menutop - menucount*16;
}
if (menubottom < 0) {
    menubottom = 0;
    menutop = menubottom + menucount*16;
}
menuleft = x - 100;
menuright = x + 100;
if (menuleft < 0) {
    menuleft = 0;
    menuright = menuleft + 200;
}
if (menuright > 1023) {
    menuright = 1023;
    menuleft = 823;
}
writemask(6);        /* restrict to menu planes */
color(2);            /* menu background */
```

```
cursoff();
rectfi(menuleft, menubottom, menuright, menutop);
color(4);              /* menu text */
move2i(menuleft, menubottom);
draw2i(menuleft, menutop);
draw2i(menuright, menutop);
draw2i(menuright, menubottom);
for (i = 0; i < menucount; i++) {
    move2i(menuleft, menutop - (i+1)*16);
    draw2i(menuright, menutop - (i+1)*16);
    cmov2i(menuleft + 10, menutop - 14 - i*16);
    charstr(names[i].text);
}
curson();
while (1) {
    x = getvaluator(MOUSEX);
    y = getvaluator(MOUSEY);
    if (menuleft < x && x < menuright && menubottom < y &&
      y < menutop) {
        highlight = (menutop - y)/16;
        cursoff();
        if (lasthighlight != -1 &&
          lasthighlight != highlight) {
            color(2);
            rectfi(menuleft+1,
                   menutop - lasthighlight*16 - 15,
                   menuright-1,
                   menutop - lasthighlight*16 - 1);
            color(4);
            cmov2i(menuleft + 10, menutop - 14 - lasthighlight*16);
            charstr(names[lasthighlight].text);
        }
        if (lasthighlight != highlight) {
            color(6);
            rectfi(menuleft+1, menutop - highlight*16 - 15,
                   menuright-1, menutop - highlight*16 - 1);
            color(4);
            cmov2i(menuleft + 10,
                   menutop - 14 - highlight*16);
            charstr(names[highlight].text);
        }
```

```
            lasthighlight = highlight;
            curson();
        } else /* the cursor is outside the menu */ {
            if (lasthighlight != -1) {
                cursoff();
                color(2);
                rectfi(menuleft+1,
                       menutop - lasthighlight*16 - 15,
                       menuright-1,
                       menutop - lasthighlight*16 - 1);
                color(4);
                cmov2i(menuleft + 10,
                       menutop - 14 - lasthighlight*16);
                charstr(names[lasthighlight].text);
                curson();
                lasthighlight = -1;
            }
        }
        if (qtest()) {
            qread(&dummy);
            qread(&x);
            qread(&y);
            color(0);
            cursoff();
            rectfi(menuleft, menubottom, menuright, menutop);
            curson();
            if (menuleft<x && x<menuright && menubottom<y &&
                  y<menutop)
                     x = (menutop - y)/16;
            else
                     x = 0;
            break;
        }
    }
    popmatrix();        /* now restore the state
                           to what the user had */
    color(savecolor);
    writemask(savemask);
    viewport(llx, urx, lly, ury);
    return names[x].type;
}
```

Here is the FORTRAN version of "popup":

```
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c       Installation note:
c     Various Fortran compilers may require different
c     styles of INCLUDEs.
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      INCLUDE 'fgl.h'
      INCLUDE 'fdevice.h'
c
      INTEGER I, J, MNAMES(7), COMAND, POPUP
      INTEGER LINE, POINTS, CIRCLE, RCTGL, RCTGLF, QUIT
      CHARACTER*20 MSTRGS(7)
      LINE = 1
      POINTS = 2
      CIRCLE = 3
      RCTGL = 4
      RCTGLF = 5
      QUIT = 6
c
c       Initialize main menu names and strings (names 1-6 are
c       as per the parameters above, but are easier to
c       init. via the DO loop)
      DO 100 I=1,6
        MNAMES(I) = I
100   CONTINUE
      MNAMES(7) = 0
c
      MSTRGS(1) = 'Line'
      MSTRGS(2) = '100 points'
      MSTRGS(3) = 'Filled circle'
      MSTRGS(4) = 'Outlined rectangle'
      MSTRGS(5) = 'Filled rectangle'
      MSTRGS(6) = 'Quit'
      MSTRGS(7) = ' '
c
      CALL GINIT()
c       background only
      CALL MAPCOL(0,0,0,0)
c       drawing only
      CALL MAPCOL(1,0,255,0)
c       popup background
```

```
      CALL MAPCOL(2,0,0,0)
C       popup background over drawing
      CALL MAPCOL(3,0,0,0)
C       popup text only
      CALL MAPCOL(4,255,255,255)
C       popup text over drawing
      CALL MAPCOL(5,255,255,255)
C       popup highlight only
      CALL MAPCOL(6,100,100,100)
C       popup highlight over drawing
      CALL MAPCOL(7,100,100,100)
      CALL SETCUR(0,4,7)
      CALL QDEVIC(LEFTMO)
      CALL TIE(LEFTMO,MOUSEX,MOUSEY)
C
      CALL CURSOF()
      CALL COLOR(0)
      CALL CLEAR()
      CALL CURSON()
      CALL VIEWPO(150,850,50,750)
      CALL ORTHO2(-1.0,1.0,-1.0,1.0)
C
C       loop until quit selected
200   CONTINUE
        COMAND = POPUP(MNAMES,MSTRGS)
        CALL CURSOF()
        CALL COLOR(0)
        CALL CLEAR()
        CALL COLOR(1)
C
        IF (COMAND .EQ. LINE) THEN
          CALL MOVE2(-1.0,-1.0)
          CALL DRAW2(1.0,1.0)
        ELSE IF (COMAND .EQ. POINTS) THEN
          DO 400 I=1,10
            DO 300 J=1,10
              CALL PNT2(REAL(I)/20.0,REAL(J)/20.0)
300         CONTINUE
400       CONTINUE
        ELSE IF (COMAND .EQ. CIRCLE) THEN
          CALL CIRCF(0.0,0.0,0.5)
```

```
          ELSE IF (COMAND .EQ. RCTGL) THEN
            CALL RECT(-0.5,-0.5,0.5,0.5)
          ELSE IF (COMAND .EQ. RCTGLF) THEN
            CALL RECTF(-0.5,-0.5,0.5,0.5)
          ELSE IF (COMAND .EQ. QUIT) THEN
            CALL GRESET()
            CALL GEXIT()
            GO TO 500
          END IF
C
          CALL CURSON()
C         loop back until quit selected
        GO TO 200
C
500     STOP
        END
C
        INTEGER FUNCTION POPUP(NAMES,STRNGS)
        INTEGER NAMES(*)
        CHARACTER*20 STRNGS(*)
        INTEGER GETCOL, GETWRI, GETVAL, QTEST
        INTEGER I, MCOUNT, MTOP, MBOTOM, MLEFT, MRIGHT
        INTEGER LASTHL, HIGHLT
        INTEGER*2 DUMMY, X, Y
        INTEGER SVCOLR, SVMASK
        INTEGER*2 LLX, LLY, URX, URY
        INTEGER ILLX, ILLY, IURX, IURY
        LASTHL = -1
C
        CALL QREAD(DUMMY)
        CALL QREAD(X)
        CALL QREAD(Y)
C         save the state of everything
        SVCOLR = GETCOL()
        SVMASK = GETWRI()
        CALL GETVIE(LLX,URX,LLY,URY)
        ILLX = LLX
        IURX = URX
        ILLY = LLY
        IURY = URY
        CALL PUSHMA()
C         now set up to draw the menu
```

```
      CALL VIEWPO(0,1023,0,767)
      CALL ORTHO2(-0.5,1023.5,-0.5,767.5)
C       set MCOUNT equal to number of names in menu
      MCOUNT = 0
1000  CONTINUE
        IF (NAMES(MCOUNT+1) .NE. 0) THEN
          MCOUNT = MCOUNT + 1
          GO TO 1000
      END IF
C
      MTOP = Y + MCOUNT * 8
      IF (MTOP .GT. 767) MTOP = 767
      MBOTOM = MTOP - MCOUNT * 16
C
      IF (MBOTOM .LT. 0) THEN
        MBOTOM = 0
        MTOP = MBOTOM + MCOUNT * 16
      END IF
C
      MLEFT = X - 100
      IF (MLEFT .LT. 0) MLEFT = 0
      MRIGHT = MLEFT + 200
C
      IF (MRIGHT .GT. 1023) THEN
        MRIGHT = 1023
        MLEFT = 823
      END IF
C
C       restrict to menu planes
      CALL WRITEM(6)
C       menu background
      CALL COLOR(2)
      CALL CURSOF()
      CALL RECTFI(MLEFT,MBOTOM,MRIGHT,MTOP)
C       menu text
      CALL COLOR(4)
      CALL RECTI(MLEFT,MBOTOM,MRIGHT,MTOP)
      DO 2000 I=1,MCOUNT
        CALL MOVE2I(MLEFT, MTOP - I*16)
        CALL DRAW2I(MRIGHT, MTOP - I*16)
        CALL CMOV2I(MLEFT + 10, MTOP - I*16 + 2)
        CALL CHARST(STRNGS(I),LEN(STRNGS(I)))
```

```
2000  CONTINUE
C
      CALL CURSON()
C       loop to highlight potential selection
C       & accept selection
3000  CONTINUE
        X = GETVAL(266)
        Y = GETVAL(267)
C
C         if the cursor is inside the menu
        IF ((MLEFT.LT.X).AND.(X.LT.MRIGHT) .AND.
     2 (MBOTOM.LT.Y).AND.(Y.LT .MTOP)) THEN
          HIGHLT = (MTOP - Y)/16
          CALL CURSOF()
C
          IF ((LASTHL.NE.-1) .AND. (LASTHL.NE.HIGHLT)) THEN
            CALL COLOR(2)
            CALL RECTFI(MLEFT+1, MTOP - LASTHL*16 - 15,
     2 MRIGHT-1, MTOP - LASTHL*16 -1)
            CALL COLOR(4)
            CALL CMOV2I(MLEFT + 10, MTOP - 14 - LASTHL*16)
            CALL CHARST(STRNGS(LASTHL+1),LEN(STRNGS(LASTHL+1)))
          END IF
C
          IF (LASTHL.NE.HIGHLT) THEN
            CALL COLOR(6)
            CALL RECTFI(MLEFT+1, MTOP - HIGHLT*16 - 15,
     2 MRIGHT-1, MTOP - HIGHLT*16 -1)
            CALL COLOR(4)
            CALL CMOV2I(MLEFT + 10, MTOP - 14 - HIGHLT*16)
            CALL CHARST(STRNGS(HIGHLT+1),LEN(STRNGS(HIGHLT+1)))
          END IF
C
          LASTHL = HIGHLT
          CALL CURSON()
C
C         if the cursor is outside the menu
        ELSE
          IF (LASTHL.NE.-1) THEN
            CALL CURSOF()
            CALL COLOR(2)
            CALL RECTFI(MLEFT+1, MTOP - LASTHL*16 - 15,
```

```
     2 MRIGHT-1, MTOP - LASTHL*16 -1)
            CALL COLOR(4)
            CALL CMOV2I(MLEFT + 10, MTOP - 14 - LASTHL*16)
            CALL CHARST(STRNGS(LASTHL+1),LEN(STRNGS(LASTHL+1)))
            CALL CURSON()
            LASTHL = -1
          END IF
        END IF
C
        IF (QTEST().NE.0) THEN
          CALL QREAD(DUMMY)
          CALL QREAD(X)
          CALL QREAD(Y)
          CALL COLOR(0)
          CALL CURSOF()
          CALL RECTFI(MLEFT,MBOTOM,MRIGHT,MTOP)
          CALL CURSON()
          IF ((MLEFT.LT.X).AND.(X.LT.MRIGHT) .AND.
     2 (MBOTOM.LT.Y).AND.(Y. LT.MTOP)) THEN
            X = (MTOP - Y)/16 + 1
          ELSE
            X = 1
          END IF
C           now restore the state to what the user had
          CALL POPMAT()
          CALL COLOR(SVCOLR)
          CALL WRITEM(SVMASK)
          CALL VIEWPO(ILLX,IURX,ILLY,IURY)
          POPUP = NAMES(X)
          GO TO 4000
        END IF
C
C       loop back to highlight potential selection & accept selecti
      GO TO 3000
C
4000  RETURN
      END
```

Example 8: Modeling Transformations

The "trans.c" program shows how the `translate()`, `rotate()`, and `scale()` commands affect a drawing. The object being drawn is a cube of side 2 centered at the origin (x = 0, y = 0, and z = 0). For reference, the letters 'X', 'Y', and 'Z' are drawn on the faces x = 1, y = 1, and z = 1, respectively. It is always viewed from a fixed point in space (x = 10.0, y = 10.0, and z = 10.0), with a fixed perspective view. Exactly one transformation at a time is applied.

The cube is drawn in a viewport that is 800 by 600 pixels — an aspect ratio of 800/600 = 1.3333333. This value must be specified to the perspective command or the picture will be squashed sideways. In almost all cases, the aspect ratio used by the perspective command will be the ratio of the length to the height of the viewport that is used.

For reference, a fixed set of axes is also drawn untransformed. The code to draw the axes is in the `drawaxes()` routine. The most important feature of this example is the way that transformations are set up and restored. The heart of the program is this:

```
<set up initial viewing transformation>
<draw everything with the current view>
<do forever> {
    <get new transformation from user (but don't apply it)>
    <draw the axes (with the original transformation)>
    pushmatrix(); /* saves and copies the current
                     transformation */
    <apply new transformation> /* this multiplies a new
                                  transformation */
                               /* by the current one */
    <draw the cube>
    popmatrix(); /* restores original transformation */
}
```

The user interface is simple — it uses only the left mouse button. When that button is pressed down, a popup menu appears. When it is released, the menu item under the cursor is selected. You can choose to translate, rotate, or scale along (or around) any of the three axes. If the button is released when the cursor is not over the menu, nothing happens, and the next press brings up the menu again. One of the menu entries is "Quit", and neither of the other buttons does anything.

Once a transformation selection is made with the menu, a control bar is drawn in the lower part of the screen. The bar is labeled with numbers that range from -10 to 10 for the translation and scaling routines, and from 0 to 3600 for the rotation commands. When the cursor is placed inside this bar, the labeled cube is drawn with the transformation selected by the menu, and with the value specified by the control bar. As long as the cursor is within the control bar, the figure will be redrawn continuously. To try a new transformation, press and release the left mouse button.

This program uses the routines in the "popuputil.c" file. The popup code is similar to that in the sample program "popup.c", so see the documentation for that program to learn how the popup menu itself works. The main difference between the code in "popup.c" and "popuputil.c" is that in "popuputil.c", four colors are reserved for drawing rather than two. The file "popup.h" defines constants for the four drawing colors. Note that the popup menu is used here in double buffer mode — it will work fine in single buffer mode too.

In many cases, popup menus in double buffer mode can be done using fewer bitplanes — especially if they are used in a dynamic scene. A menu in this mode does not need its own bitplanes since the scene is regenerated for each frame. The menu is simply drawn on top of the scene each time.

The code for the main routine is fairly simple. It gets a command and sets up the appropriate control bar. Then it goes into a loop that tests for a button press and if there is none, reads the control bar, performs the appropriate transformation, and goes back to test the event queue. Note that `pushmatrix()` and `popmatrix()` surround the transformation so that the transformation is done relative to the original viewing setup. If this were omitted, the transformations would be compounded (with astonishing results).

Note also that the transformation parameter has to be cast to an integer when it is used in a rotation command. As usual, all angles are in tenths of degrees, which is why the control bar runs from 0 to 3600.

The source for "trans.c" follows:

```
/* "trans.c" */
#include "gl.h"
#include "device.h"
#include "popup.h"

#define CUBEOBJ 1
#define AXISOBJ 2

#define TRANSX 1
#define TRANSY 2
#define TRANSZ 3
#define ROTX 4
#define ROTY 5
#define ROTZ 6
#define SCALEX 7
#define SCALEY 8
#define SCALEZ 9
#define EXITTRANS 10

popupentry mainmenu[] = {
    TRANSX, "Translate X",
    TRANSY, "Translate Y",
    TRANSZ, "Translate Z",
    ROTX, "Rotate X",
    ROTY, "Rotate Y",
    ROTZ, "Rotate Z",
    SCALEX, "Scale X",
    SCALEY, "Scale Y",
    SCALEZ, "Scale Z",
    EXITTRANS, "Quit",
    0, 0
};

main()
{
```

```
Device val;
int command, i;
float transparam;

ginit();
doublebuffer();
gconfig();
initpopup();
frontbuffer(TRUE);
writemask(0xfff);
color(BLACKDRAW);
cursoff();
clear();
curson();
frontbuffer(FALSE);
viewport(100, 900, 100, 700);
perspective(400, 1.3333333, 0.1, 1000.0);
lookat(10.0, 10.0, 10.0, 0.0, 0.0, 0.0, 0);

while (1) {
    frontbuffer(TRUE); /* make the menu visible */
    switch (command = popup(mainmenu)) {
        case 0:
            continue;
        case TRANSX:
        case TRANSY:
        case TRANSZ:
        case SCALEX:
        case SCALEY:
        case SCALEZ:
            drawbar(-10.0, 10.0);
            break;
        case ROTX:
        case ROTY:
        case ROTZ:
            drawbar(0.0, 3600.0);
            break;
        case EXITTRANS:
            greset();
            gexit();
            exit(0);
```

```
}
frontbuffer(FALSE);
color(BLACKDRAW);
clear();
color(YELLOWDRAW);
drawaxes();
color(REDDRAW);
drawcube();
swapbuffers();
gflush();
while (qtest() == 0)    /* while no buttons pressed */ {
    if (readbar(&transparam)) {
         color(BLACKDRAW);
         clear();
         color(YELLOWDRAW);
         drawaxes();
         pushmatrix();
         switch (command) {
    case TRANSX:
        translate(transparam, 0.0, 0.0);
        break;
    case TRANSY:
        translate(0.0, transparam, 0.0);
        break;
    case TRANSZ:
        translate(0.0, 0.0, transparam);
        break;
    case SCALEX:
        scale(transparam, 1.0, 1.0);
        break;
    case SCALEY:
        scale(1.0, transparam, 1.0);
        break;
    case SCALEZ:
        scale(1.0, 1.0, transparam);
        break;
    case ROTX:
        rotate((int)transparam, 'x');
        break;
    case ROTY:
        rotate((int)transparam, 'y');
```

```
                break;
            case ROTZ:
                rotate((int)transparam, 'z');
                break;
                }
                color(REDDRAW);
                drawcube();
                popmatrix();
                swapbuffers();
                gflush();
            }
        }
        for (i = 0; i < 6; i++)
            qread(&val);/* throw away down and up strokes */
              /* of the exit button press */
    }
}

float barmin, barmax, bardelta;

drawbar(minval, maxval)
float minval, maxval;
{
    register i;
    char str[20];

    barmin = minval;
    barmax = maxval;
    bardelta = (barmax - barmin)/800.0;
    pushmatrix();
    pushviewport();
    ortho2(-0.5, 1023.5, -0.5, 767.5);
    viewport(0, 1023, 0, 767);
    frontbuffer(TRUE);
    cursoff();
    color(BLACKDRAW);
    rectfi(99, 19, 1000, 70);
    color(REDDRAW);
    recti(100, 20, 900, 40);
    for (i = 0; i < 5; i++) {
        move2i(100 + i*200, 40);
```

```
        draw2i(100 + i*200, 50);
        cmov2i(103 + i*200, 44);
        sprintf(str, "%6.2f", minval + i*(maxval - minval)/4.0);
        charstr(str);
    }
    curson();
    frontbuffer(FALSE);
    popviewport();
    popmatrix();
}

/* The readbar routine returns 1 if the value stored in
 * retval is valid, and zero otherwise.
 */

readbar(retval)
float *retval;
{
    int xmouse, ymouse;

    ymouse = getvaluator(MOUSEY);
    if (20 <= ymouse && ymouse <= 40) {
        xmouse = getvaluator(MOUSEX);
        if (100 <= xmouse && xmouse <= 900) {
            *retval = barmin + bardelta*(xmouse - 100);
            return 1;
        }
    }
    return 0;
}

drawcube()
{
    static short initialized = 0;

    if (!initialized) {
        makeobj(CUBEOBJ);

        /* First draw the outline of the cube */

        move(-1.0, -1.0, -1.0);
```

```
        draw(1.0, -1.0, -1.0);
        draw(1.0, 1.0, -1.0);
        draw(-1.0, 1.0, -1.0);
        draw(-1.0, -1.0, -1.0);
        draw(-1.0, -1.0, 1.0);
        draw(1.0, -1.0, 1.0);
        draw(1.0, 1.0, 1.0);
        draw(-1.0, 1.0, 1.0);
        draw(-1.0, -1.0, 1.0);
        move(-1.0, 1.0, -1.0);
        draw(-1.0, 1.0, 1.0);
        move(1.0, 1.0, -1.0);
        draw(1.0, 1.0, 1.0);
        move(1.0, -1.0, 1.0);
        draw(1.0, -1.0, -1.0);

        /*now draw the letters 'X', 'Y', and 'Z' on the faces:*/

        move(1.0, -0.6666666, -0.5);
        draw(1.0, 0.6666666, 0.5);
        move(1.0, 0.6666666, -0.5);
        draw(1.0, -0.6666666, 0.5);

        move(0.0, 1.0, 0.6666666);
        draw(0.0, 1.0, 0.0);
        draw(0.5, 1.0, -0.6666666);
        move(0.0, 1.0, 0.0);
        draw(-0.5, 1.0, -0.6666666);

        move(-0.5, 0.6666666, 1.0);
        draw(0.5, 0.6666666, 1.0);
        draw(-0.5, -0.6666666, 1.0);
        draw(0.5, -0.6666666, 1.0);
        closeobj();
        initialized = 1;
    }
    callobj(CUBEOBJ);
    gflush();
}

drawaxes()
```

```
{
    static initialized = 0;

    if (!initialized) {
        makeobj(AXISOBJ);
        movei(0, 0, 0);
        drawi(0, 0, 2);
        movei(0, 2, 0);
        drawi(0, 0, 0);
        drawi(2, 0, 0);
        cmovi(0, 0, 2);
        charstr("z");
        cmovi(0, 2, 0);
        charstr("y");
        cmovi(2, 0, 0);
        charstr("x");
        closeobj();
        initialized = 1;
    }
    callobj(AXISOBJ);
    gflush();
}
```

The source for "popup.h" follows:

```
/* "popup.h" */
#ifndef POPUPHEADER
#define POPUPHEADER

#define BLACKDRAW       0
#define GREENDRAW       1
#define REDDRAW         2
#define YELLOWDRAW      3

typedef struct {
    short type;
    char *text;
} popupentry;

#endif
```

The source for "popuputil.c" follows:

```
/* "popuputil.c" */
#include "gl.h"
#include "device.h"
#include "popup.h"

initpopup()
{
    register i;

    mapcolor(BLACKDRAW, 0, 0, 0);        /* background only */
    mapcolor(GREENDRAW, 0, 255, 0);      /* green drawing */
    mapcolor(REDDRAW, 255, 0, 0);        /* red drawing */
    mapcolor(YELLOWDRAW, 255, 255, 0);  /* yellow drawing */
    for (i = 4; i < 8; i++)
        mapcolor(i, 0, 0, 0);     /* popup background */
    for (i = 8; i < 12; i++)
        mapcolor(i, 255, 255, 255);   /* popup text */
    for (i = 12; i < 16; i++)
        mapcolor(i, 150, 150, 150);   /* popup highlight */
    setcursor(0, 8, 15);
    qdevice(LEFTMOUSE);
    tie(LEFTMOUSE, MOUSEX, MOUSEY);
}

popup(names)
popupentry names[];
{
    register short i, menucount;
    short menutop, menubottom, menuleft, menuright;
    short lasthighlight = -1, highlight;
    Device dummy, x, y;
    short savecolor, savemask;

    menucount = 0;
    qread(&dummy);
    qread(&x);
    qread(&y);
    pushattributes();
    pushviewport();
```

```
pushmatrix();
viewport(0, 1023, 0, 767);
ortho2(-0.5, 1023.5, -0.5, 767.5);
while (names[menucount].type)
     menucount++;
menutop = y + menucount*8;
menubottom = y - menucount*8;
if (menutop > 767) {
     menutop = 767;
     menubottom = menutop - menucount*16;
}
if (menubottom < 0) {
     menubottom = 0;
     menutop = menubottom + menucount*16;
}
menuleft = x - 100;
menuright = x + 100;
if (menuleft < 0) {
     menuleft = 0;
     menuright = menuleft + 200;
}
if (menuright > 1023) {
     menuright = 1023;
     menuleft = 823;
}
writemask(12);       /* restrict to menu planes */
color(4);            /* menu background */
cursoff();
rectfi(menuleft, menubottom, menuright, menutop);
color(8);            /* menu text */
move2i(menuleft, menubottom);
draw2i(menuleft, menutop);
draw2i(menuright, menutop);
draw2i(menuright, menubottom);
for (i = 0; i < menucount; i++) {
     move2i(menuleft, menutop - (i+1)*16);
     draw2i(menuright, menutop - (i+1)*16);
     cmov2i(menuleft + 10, menutop - 14 - i*16);
     charstr(names[i].text);
}
curson();
```

```
while (1) {
  x = getvaluator(MOUSEX);
  y = getvaluator(MOUSEY);
  if (menuleft < x && x < menuright && menubottom < y && y < menutop
        highlight = (menutop - y)/16;
        cursoff();
        if (lasthighlight != -1 && lasthighlight != highlight) {
           color(4);
           rectfi(menuleft+1, menutop - lasthighlight*16 - 15,
              menuright-1, menutop - lasthighlight*16 - 1);
           color(8);
           cmov2i(menuleft + 10, menutop - 14 - lasthighlight*16);
           charstr(names[lasthighlight].text);
        }
        if (lasthighlight != highlight) {
           color(12);
           rectfi(menuleft+1, menutop - highlight*16 - 15,
              menuright-1, menutop - highlight*16 - 1);
           color(8);
           cmov2i(menuleft + 10, menutop - 14 - highlight*16);
           charstr(names[highlight].text);
        }
        lasthighlight = highlight;
        curson();
    } else /* the cursor is outside the menu */ {
        if (lasthighlight != -1) {
           cursoff();
           color(4);
           rectfi(menuleft+1, menutop - lasthighlight*16 - 15,
              menuright-1, menutop - lasthighlight*16 - 1);
           color(8);
           cmov2i(menuleft + 10, menutop - 14 - lasthighlight*16);
           charstr(names[lasthighlight].text);
           curson();
           lasthighlight = -1;
        }
    }
    if (qtest()) {
        qread(&dummy);
        qread(&x);
        qread(&y);
```

```
        color(0);
        cursoff();
        rectfi(menuleft, menubottom, menuright, menutop);
        curson();
        if (menuleft<x && x<menuright && menubottom<y && y<menutop)
            x = (menutop - y)/16;
        else
            x = 0;
        break;
        }
    }
    popmatrix();
    popviewport();
    popattributes();
    return names[x].type;
}
```

Example 9: Writemasks and Color Maps

The sample program "vlsi.c" illustrates the use of the writemask in conjunction with the color map. It happens to be a simple-minded VLSI circuit editor, although the principles illustrated would be useful in many applications. Even if you have no interest in VLSI, it is worthwhile to study this example until you understand it thoroughly — the tricks used here can be used in many applications.

The problem the program solves is this: an integrated circuit chip has many layers on it — metal (blue), polysilicon (red), diffusion (green), contact cuts (black), and others. The meanings of these layers do not matter here — the point is that the circuit designer would like to see what layers are present at any point on the screen. When red crosses green, for example, a transistor is formed, and where both green and red appear, it would be nice to show it as a brillant yellow. In most cases, it would also be nice to be able to look at the color and know exactly what layers are there.

The contact cut (black) is different — after the other layers have been put down, a contact cut removes everything that was there, so it might as well be shown as black, no matter what layers are present.

The way this sample editor will work is to draw each of the layers on a separate bitplane, so each pixel on the screen may be covered by any combination of the four layers mentioned above. Colors are then assigned to the layer patterns. In this example, the following color assignments are made:

BP3 (cut)	BP2 (poly)	BP1 (diff)	BP0 (metal)	RED	GREEN	BLUE	
0	0	0	0	255	255	255	(white - nothing there)
0	0	0	X	0	0	255	(metal only - show blue)
0	0	X	0	0	255	0	(diffusion only - green)
0	0	X	X	0	150	255	(metal + diff - purple)
0	X	0	0	255	0	0	(polysilicon only - red)
0	X	0	X	150	0	255	(light blue)
0	X	X	0	255	255	0	(transistor - yellow)
0	X	X	X	150	100	0	(brown)
X	0	0	0	0	0	0	(contact cut present)
X	0	0	X	0	0	0	(")
X	0	X	0	0	0	0	(")
X	0	X	X	0	0	0	(")
X	X	0	0	0	0	0	(")
X	X	0	X	0	0	0	(")
X	X	X	0	0	0	0	(")
X	X	X	X	0	0	0	(")

The colors chosen to represent the combinations are arbitrary. In the bitplane columns (BP0, ...) above, X means that the substance is present; 0 means it is absent.

If we replace all of the X's in the bitplane columns above, we can read off each 4-bit pattern as a binary number. Think of BP0 as the 1's place, BP1 as the 2's place, BP2 as the 4's place, and BP3, BP4, BP5, ... as the 8's, 16's, 32's, ... place. Add up the numbers to find out the representation. For example, "X 0 X 0" (BP3 and BP1) above corresponds to the number 8+2 = 10. We would map this color with `mapcolor(10,0,0,0)`.

Notice that in the program a fifth plane (BP4) is also used. The cursor will be drawn in that plane, and the color maps are rigged so that the cursor will appear to be white if it is over a black area, and black otherwise. This way, the cursor will always show up.

The source for "vlsi.c" appears below:

```
/* "vlsi.c" */
#include "gl.h"
#include "device.h"

main()
{
    register i;
    Device dummy, xend, yend, xstart, ystart, type;
    short wm;

    ginit();
    mapcolor(0, 255, 255, 255); /* WHITE */
    mapcolor(1, 0, 0, 255);     /* BLUE */
    mapcolor(2, 0, 255, 0);     /* RED */
    mapcolor(3, 0, 150, 255);   /* PURPLE */
    mapcolor(4, 255, 0, 0);     /* GREEN */
    mapcolor(5, 150, 0, 255);   /* LIGHT BLUE */
    mapcolor(6, 255, 255, 0);   /* YELLOW */
    mapcolor(7, 150, 100, 0);   /* BROWN */
    for (i = 8; i < 24; i++)
        mapcolor(i, 0, 0, 0);   /* BLACK */
    for (i = 24; i < 32; i++)
        mapcolor(i, 255, 255, 255);     /* WHITE */
    qdevice(LEFTMOUSE);
    tie(LEFTMOUSE, MOUSEX, MOUSEY);
    qdevice(MIDDLEMOUSE);
    tie(MIDDLEMOUSE, MOUSEX, MOUSEY);
    qdevice(RIGHTMOUSE);
    qdevice(KEYBD);
    setcursor(0, 16, 16);
    restart();
    while (1)
        switch (type = qread(&dummy)) {
            case KEYBD:
                gexit();
                exit(0);
            case RIGHTMOUSE:
                qread(&dummy);
                restart();
```

```
                break;
            case MIDDLEMOUSE:
            case LEFTMOUSE:
                qread(&xstart);
                qread(&ystart);
                if (xstart < 60) {
                    if (10 <= xstart && xstart <= 50) {
                        if (10 <= ystart && ystart <= 50)
                            wm = 1;
                        else if (60 <= ystart && ystart <= 100)
                            wm = 2;
                        else if (110 <= ystart && ystart <= 150)
                            wm = 4;
                        else if (160 <= ystart && ystart <= 200)
                            wm = 8;
                        writemask(wm);
                        qread(&dummy);
                        qread(&dummy);
                        qread(&dummy);
                    }
                } else {
                    qread(&dummy);
                    qread(&xend);
                    qread(&yend);
                    if (xend > 60) {
                        cursoff();
                        if (type == LEFTMOUSE)
                            color(31);  /* draw */
                        else
                            color(0);   /* erase */
                        rectfi(xstart, ystart, xend, yend);
                        curson();
                        gflush();
                    }
                }
        }
}

restart()
{
    writemask(0xfff);
```

```
    cursoff();
    color(0);
    clear();
    color(1);
    rectfi(10, 10, 50, 50);
    color(2);
    rectfi(10, 60, 50, 100);
    color(4);
    rectfi(10, 110, 50, 150);
    color(8);
    rectfi(10, 160, 50, 200);
    move2i(60, 0);
    draw2i(60, 767);
    color(31);
    writemask(0);
    curson();
}
```

The user interface is quite simple — typing any keyboard character will cause the program to halt, and the right mouse button will clear the screen for a fresh start. The left and middle mouse buttons are draw and erase, respectively. The rectangle swept out between the time a button goes down and when it comes up is either drawn or erased. To draw a rectangle, press down the left mouse button at one corner, hold it down, and move the mouse to the opposite corner. When the button is released, the rectangle is drawn on the current layer.

The draw and erase buttons only work in the right part of the screen. If either of those buttons is pressed in the menu area (the leftmost 60 pixels on the screen) within one of the four menu regions, that color becomes the current layer.

The writemask is used to control which bit planes will be written into. In this example, except during the initialization and reinitialization, at most one bitplane is enabled for writing at a time — the writemask is set to 1, 2, 4, or 8. Once the writemask is set, the drawing code does not need to know its setting — it simply needs to know whether it is drawing or erasing. If it is drawing, a 1 must be written into the enabled plane; if it is erasing, a 0 must be written. Thus, to draw, set the color to 31 (= 16 + 8 + 4 + 2 + 1 — all planes). The writemask will only let one plane get written. To erase, set the color to 0.

Basically, the writemask tells which planes are to be changed, and for all those planes with writemask 1, the color tells what to write in as the new value.

Example 10: A Color Editor

The sample program "colored.c" is a very simple color editor. All the colors that the IRIS can display are combinations of various amounts of red, green, and blue components. Each of the three components can vary between 0 and 255. If red = green = blue = 0, the color is black; red = green = blue = 255 gives white. If red = 255 and green = blue = 0, we get the color red. Altogether, there are 256*256*256 = 16,777,216 such colors.

An IRIS in RGB mode with 24 bitplanes can display any combination of these colors; if there are fewer than 24 bitplanes, or if the IRIS is in single or double buffer mode, there are at most 4095 different colors available, and they are defined by a color map. Every entry in the map specifies an amount of red, green, and blue to use. The command `mapcolor(17,255,150,0)` defines color 17 to have a red component of 255, a green component of 150, a blue component of zero — a light orange color. `color(17)` would then set the current color to that shade of orange. If anything on the screen was painted in color 17, it would also change to orange.

`ginit()` initializes all of the color map entries to black except for seven of them. Effectively, `ginit()` makes the following calls on `mapcolor()`:

```
for (i = 0; i < 4095; i++)
    mapcolor(i, 0, 0, 0);      /* BLACK */
mapcolor (1,255,0,0);          /* RED */
mapcolor (2,0,255,0);          /* GREEN */
mapcolor (3,255,255,0);        /* YELLOW */
mapcolor (4,0,0,255);          /* BLUE */
mapcolor (5,255,0,255);        /* MAGENTA */
mapcolor (6,0,255,255);        /* CYAN */
mapcolor (7,255,255,255);      /* WHITE */
```

The file "device.h" defines RED to be 1, GREEN to be 2, and so on. The definitions of any of these colors can be reset, as well as any of the other entries in the color map.

The "colored.c" program makes it possible to play with the red, green, and blue components to see how colors are affected by them. It always has a current color that is displayed in a large rectangle at the bottom of the screen. Above this rectangle, the numeric values of the red, green, and blue components are displayed. There are also three color bars (one for red, one for green, and one for blue). Along the red color bar are series of colors where the blue and green components are fixed (to those of the current color), but the red component varies through the range 0 to 255. The blue bar is the same, but with the blue component varying, and so on.

If the cursor is moved into one of the color bars (or slightly above it) and pressed, that color becomes the current color, and the shades on the other three color bars are recomputed. This makes it easy to zero in on a new color. This program is designed to work on an IRIS with eight bitplanes (the minimum configuration) which can only use 256 different colors, so only 64 shades are shown on each color bar. With minor modifications, the program could be made to show all 256 colors if the IRIS has twelve or more bitplanes.

The code for "colored.c" follows:

```
/* "colored.c" */
#include "gl.h"
#include "device.h"

#define MYBLACK 255
#define MYWHITE 254
#define CURRENTCOLOR 253

#define indextovalue(index) (4*index + 3)

short redindex = 0, greenindex = 0, blueindex = 0;

main()
{
    register i, j;
    Device dummy, xpos, ypos;
```

```
ginit();
color(0);
writemask(0xfff);          /* get zeroes in all planes */
clear();
mapcolor(MYBLACK, 0, 0, 0);          /* black */
mapcolor(MYWHITE, 255, 255, 255);    /* white */
mapcolor(CURRENTCOLOR, 0, 0, 0);
qdevice(LEFTMOUSE);
tie(LEFTMOUSE, MOUSEX, MOUSEY);
qdevice(RIGHTMOUSE);
setcursor(0, MYWHITE, 0xfff);
buildmap();
displaymap();
while (1) {
    j = -1;
    switch (qread(&dummy)) {
        case RIGHTMOUSE:
            greset();
            gexit();
            exit(0);
        case LEFTMOUSE:
            qread(&xpos);
            qread(&ypos);
            qread(&dummy);
            qread(&dummy);
            qread(&dummy);
            if (650 <= ypos && ypos <= 720)
                i = 0;       /* red color bar */
            else if (550 <= ypos && ypos <= 620)
                i = 1;       /* green color bar */
            else if (450 <= ypos && ypos <= 520)
                i = 2;       /* blue color bar */
            else i = -1;
            if (i != -1) {
                if (200 <= xpos && xpos < 840)
                    j = (xpos - 200)/10;
            }
            if (j != -1) {  /* valid input event */
                switch (i) {
                    case 0:
                        redindex = j;
```

```
                                        break;
                                    case 1:
                                        greenindex = j;
                                        break;
                                    case 2:
                                        blueindex = j;
                                        break;
                                }
                                buildmap();
                                displaymap();
                            }
                }
        }
}

buildmap()
{
        register i, j;

        blankscreen(TRUE);
        for (i = 0; i < 3; i++)
            for (j = 0; j < 64; j++)
                switch (i) {
                    case 0:        /* red */
                        mapcolor(i*64+j, indextovalue(j),
                            indextovalue(greenindex),
                            indextovalue(blueindex));
                        break;
                    case 1: /* green */
                        mapcolor(i*64+j, indextovalue(redindex),
                             indextovalue(j),
                             indextovalue(blueindex));
                        break;
                    case 2: /* blue */
                        mapcolor(i*64+j, indextovalue(redindex),
                            indextovalue(greenindex),
                            indextovalue(j));
                        break;
                }
        mapcolor(CURRENTCOLOR, indextovalue(redindex),
                    indextovalue(greenindex),
```

```
                    indextovalue(blueindex));
    blankscreen(FALSE);
}

displaymap()
{
    register i, j;
    static initialized = 0;
    char redstr[10], greenstr[10], bluestr[10];

    if (!initialized)
        {
            makeobj(1);
            color(MYBLACK);
            clear();
            for (i = 0; i < 3; i++)
            for (j = 0; j < 64; j++) {
            for(i*64 + j);
              rectfi(200 + 10*j, 700 - i*100, 210 + 10*j,
                650 - i*100);
              color(MYWHITE);
              recti(200 + 10*j, 700 - i*100, 210 + 10*j,
                650 - i*100);
                          }
            color(CURRENTCOLOR);
            rectfi(400, 200, 600, 300);
            color(MYWHITE);
            recti(400, 200, 600, 300);
            cmov2i(150, 670);
            charstr("RED");
            cmov2i(150, 570);
            charstr("GREEN");
            cmov2i(150, 470);
            charstr("BLUE");
            cmov2i(275, 245);
            charstr("CURRENT COLOR");
            cmov2i(380, 100);
            charstr("Left mouse button: choose a color");
            cmov2i(380, 84);
            charstr("Right mouse button: exit");
```

```
            closeobj();
            initialized = 1;
        }
    cursoff();
    callobj(1);
    move2i(205 + 10*redindex, 700);
    draw2i(205 + 10*redindex, 720);
    cmov2i(210 + 10*redindex, 705);
    sprintf(redstr, "%d", indextovalue(redindex));
    charstr(redstr);
    move2i(205 + 10*greenindex, 600);
    draw2i(205 + 10*greenindex, 620);
    cmov2i(210 + 10*greenindex, 605);
    sprintf(greenstr, "%d", indextovalue(greenindex));
    charstr(greenstr);
    move2i(205 + 10*blueindex, 500);
    draw2i(205 + 10*blueindex, 520);
    cmov2i(210 + 10*blueindex, 505);
    sprintf(bluestr, "%d", indextovalue(blueindex));
    charstr(bluestr);
    cmov2i(450, 310);
    charstr("(");
    charstr(redstr);
    charstr(", ");
    charstr(greenstr);
    charstr(", ");
    charstr(bluestr);
    charstr(")");
    curson();
    gflush();
}
```

The program uses colors 0 through 63 for the 64 shades on the red color bar; colors 64 through 127 for the green bar, and 128 through 191 for the blue bar. When the current color is changed, 192 new calls on `mapcolor()` are made to change the definitions of colors 1 through 191. To guarantee that the text and background are white and black, respectively, one of the first things done in the main program is to define color 255 to be black and color 254 to be white. These definitions never change. For convenience, color 253 is set to the current color (initially black).

The left mouse button selects new current colors, and the right mouse quits. Both of these buttons are queued, and the left mouse button is tied to the mouse coordinates. The cursor is set to use the color 254 (guaranteed white), and the values of the color bars are computed by `buildmap()`, and everything is displayed by the `displaymap()` command. From then on, the program sits in a loop waiting for an input event. If the left mouse button is pressed, the code in `main()` checks to see if it is within any of the color bars. If it is, the current red, green, or blue component is reset, the color map is rebuilt, and everything is redrawn.

The `buildmap()` procedure just issues a series of `mapcolor()` commands. Its only interesting feature is the `blankscreen(TRUE)` and `blankscreen(FALSE)` commands surrounding the `mapcolor()` commands. When many color map entries are changed, there may be glitches on the screen, and the `blankscreen()` command turns off the display while the changes are being made.

`displaymap()` redraws the screen, including all of the rectangles in the color bars, the current color rectangle, the text, and the little markers above each bar that show the current setting. Almost everything is the same in each redrawing. Color 0 is always drawn in the first rectangle of the red bar, color 1 in the second rectangle, and so on. (The definition of colors 0 and 1 will have changed, but the commands to display them are unchanged.) In fact, only a few things change — the positions and labels on the color bar current position markers, and the red, green, and blue components of the current color. To make redrawing faster, all the drawing commands that are constant are compiled into a display list object (object 1), and are redrawn with the single command `callobj(1)`. The very first time `displaymap()` is called, this object will not have been created, so the static variable `initialized` is checked before each drawing.

The easiest way to understand what the code in `displaymap()` is doing is to run the program and compare the picture on the screen with a listing of the `displaymap()` code.

Workshop 1: diamond1.c

```
/*  diamond1.c  --  This program draws a static image of a baseball     */
/*                  diamond.  From this simple starting block, a        */
/*                  more sophisticated series of programs with color,   */
/*                  motion, user interaction and 3-D will be built.     */

/*  Every graphics program which utilizes the IRIS graphics library     */
/*  should include the gl.h and device.h files.                         */

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

/*  For each program in the series, new graphics library routines       */
/*  will be used.  Each new routine will be introduced in a short       */
/*  header comment.  For more information on each routine, see the      */
/*  Reference Manual section of the User's Guide.                       */
/*                                                                      */
/*  graphics library calls introduced:                                  */
/*      keepaspect (x, y) -- constrain any window opened in the future  */
/*              to an aspect ratio of y divided by x units              */
/*      prefsize (x, y) -- constrain any window opened in the future to */
/*              a size of x units by y units                            */
/*      winopen ("diamond") -- open a window, name it "diamond"         */
/*      color (colorname) -- change the current color to colorname      */
/*      clear () -- clear every pixel to the current color              */
/*      arcs (x, y, rad, startang, endang) -- draw an UNfilled arc      */
/*              with a center at (x, y), a radius of length rad.        */
/*              The arc begins at angle startang and ends at endang,    */
/*              where startang and endang are measured in tenths of     */
/*              degrees, counterclockwise from the x-axis.              */
/*              arcs is different from arc, because arcs specifies      */
/*              the x, y and rad arguments to be 16-bit integers        */
/*              (Screen coordinates) rather than 32-bit floating point  */
/*              real numbers, which are used for arc.  arcf is used     */
/*              for filled arcs.                                        */
/*      move2s (x, y) -- move, without drawing, the current graphics    */
/*              position to the point (x, y).  move2 (x, y) without     */
/*              the s means to move to an (x, y) location specified     */
/*              by 32-bit real numbers.  move (x, y, z) without the 2   */
/*              means to move to a 3-D location.  Other calls in the    */
```

```
/*              move family are move2i (2-D integer), moves (3-D short  */
/*              integer), movei (3-D integer).                           */
/*      draw2s (x, y) -- draws a line from the current graphics          */
/*              position (specified by move) to (x, y).  The current     */
/*              graphics position is then moved to (x, y).  draw2s       */
/*              also has its sibling calls:  draw, draw2, drawi,         */
/*              draw2i, draws.                                           */
/*      circfs (x, y, rad) -- draw a FILLED circle with a center         */
/*              at (x, y) and a radius of length rad.  An unfilled       */
/*              (hollow) circle would be drawn with the circ, circi,     */
/*              and circs commands.                                      */
/*      pmv2s (x, y)                                                     */
/*      pdr2s (x, y)                                                     */
/*      pclos () -- The pmv, pdr and pclos are used in conjunction       */
/*              with one another to draw a FILLED polygon.  The          */
/*              location of the first point is determined by pmv2s       */
/*              (polygon move).  The location of the next points         */
/*              are determined by a sequence of pdr2s's.  The polygon    */
/*              is closed by pclos, which connects the final point       */
/*              (last pdr) with the first point (pmv).                   */
/*              Also see the polf command for another method of          */
/*              drawing filled polygons.                                 */
/*              A concave polygon will not look right.                   */
/*      rpdr2s (rx, ry) -- relative polygon draw                         */
/*              rpdr is similar to pdr, except that the edge of a        */
/*              polygon is drawn from the current graphics position      */
/*              to a point a distance away.  You are no longer           */
/*              specifying the point you are drawing to, but how         */
/*              far away to draw FROM your current position.             */
/*                                                                       */
/*  The following calls are used to handle input from the window         */
/*  manager and other input sources. They will be discussed in gory      */
/*  detail in the queue7 workshop.                                       */
/*      qdevice () -- Establish input device.                            */
/*      qtest () -- check input queue for any input.                     */
/*      qread () -- Read data from the input queue.                      */
/*  Some lines are drawn twice their previous thickness with the         */
/*  linewidth() command.                                                 */

/*  In the main routine of the program, initialize() is called to        */
/*  open a window.  Then, drawimage() draws the baseball diamond.        */
/*  The while(TRUE) loop keeps the window open forever.  The user        */
/*  can kill the program from the window manager menu.                   */

main () {
    initialize ();
    drawimage ();
    while (TRUE) {      /*  loop forever (killed from window manager).  */
        while (!qtest())/*  wait for input                               */
            ;
        processinput(); /*  process input                                */
    }
}
```

```
/*  Open the window.  The prefsize() call forbids changing the window   */
/*  size from the size designated.  The qdevice call establishes         */
/*  contact with the window manager in regard to screen refreshes.       */

initialize () {

    prefsize (450, 450);
    winopen ("diamond");

    qdevice( REDRAW );
}

/*  Process input from the window manager                                */

processinput() {
    short val;
    int dev;

    while (qtest()) {                /* while input on queue.            */
        dev = qread(&val);           /* read input device data.          */
        switch (dev) {
        case REDRAW:                 /* window manager asking for redraw.*/
            reshapeviewport();       /* This shapes the window.          */
            drawimage();             /* This redraws the image.          */
            break;
        default:
            break;
        }
    }
}

/*  clear the window to BLACK.  Draw the baseball diamond.               */

drawimage () {
    color (BLACK);
    clear ();
    diamond ();
}

/*  Draw the baseball field in yellow.  Use arcs, circles, lines         */
/*  and polygons.  Note that we are drawing to a 2-D screen and all      */
/*  values are integers, so far.                                         */

diamond () {
    color (YELLOW);
    linewidth(2);         /*  change thickness of lines    */
    arcs (0, 0, 375, 0, 900);   /*  fences                 */
    arcs (0, 0, 150, 0, 900);   /*  infield                */

    move2s (0, 0);              /*  foul lines             */
    draw2s (0, 400);
    move2s (0, 0);
    draw2s (400, 0);
```

```
    linewidth(1);        /*  restore thickness of lines   */

    circfs (43, 43, 10);         /*  pitcher's mound     */

    draw_base (90, 0);           /*  first, second and third bases       */
    draw_base (90, 90);
    draw_base (0, 90);

    pmv2s (0, 0);                /*  draw home plate     */
    pdr2s (0, 3);
    pdr2s (3, 6);
    pdr2s (6, 3);
    pdr2s (3, 0);
    pclos ();
}

/*  Draw a base.  Note the use of relative draws to create the base.    */
/*  If absolute draws were used, the code would look like this:         */
/*      pmv2s (x, y);                                                   */
/*      pdr2s (x + 5, y);                                               */
/*      pdr2s (x + 5, y + 5);                                           */
/*      pdr2s (x, y + 5);                                               */
/*      pclos ();                                                       */

draw_base (x, y)
short   x,
        y;
{
    pmv2s (x, y);
    rpdr2s (5, 0);
    rpdr2s (0, 5);
    rpdr2s (-5, 0);
    pclos ();
}
```

Workshop 2:   color2.c

```
/*  color2.c  -- This program draws a static image of a baseball          */
/*               diamond with several colors.  The field is covered       */
/*               with grass.                                              */
/*  graphics library calls introduced:  mapcolor(index, r, g, b)          */

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

/*  The main routine has not changed since the last example.              */

main () {
    initialize ();
    drawimage ();
    while (TRUE) {       /*  loop forever (killed from window manager).  */
        while (!qtest())/*  wait for input                                */
            ;
        processinput(); /*  process input                                 */
    }
}

/*  Initialization now includes defining colors 8 and 9.                  */

initialize () {

    prefsize (450, 450);
    winopen ("diamond");

    qdevice( REDRAW );

    mapcolor (8, 240, 240, 240);/*  make color of fences, lines */
    mapcolor (9, 0, 175, 0);    /*  make color of grass         */
}

/*  Process input from the window manager                                 */

processinput() {
    short val;
    int dev;
```

```
    while (qtest()) {                   /* while input on queue.          */
        dev = qread(&val);              /* read input device data.        */
        switch (dev) {
        case REDRAW:                    /* window manager asking for redraw.*/
            reshapeviewport();          /* This shapes the window.        */
            drawimage();                /* This redraws the image.        */
            break;
        default:
            break;
        }
    }
}

/*  drawimage() has not changed since the last example.                   */

drawimage () {
    color (BLACK);
    clear ();
    diamond ();
}

/*  Declare color 9 to be the current color.  Then draw a filled arc      */
/*  (in color 9) for the grass of the entire field.  Draw remaining       */
/*  field features with color 8.                                          */

diamond () {
    color (9);           /*  make color 9 current color  */
    arcfs (0, 0, 375, 0, 900);  /*  grass                 */
    color (8);           /*  make color 8 current color  */
    linewidth(2);        /*  change thickness of lines   */
    arcs (0, 0, 375, 0, 900);   /*  fences                */
    arcs (0, 0, 150, 0, 900);   /*  infield               */

    move2s (0, 0);              /*  foul lines            */
    draw2s (0, 400);
    move2s (0, 0);
    draw2s (400, 0);
    linewidth(1);        /*  restore thickness of lines  */

    circfs (43, 43, 10);        /*  pitcher's mound       */

    draw_base (90, 0);          /*  first, second and third bases         */
    draw_base (90, 90);
    draw_base (0, 90);

    pmv2s (0, 0);               /*  draw home plate       */
    pdr2s (0, 3);
    pdr2s (3, 6);
    pdr2s (6, 3);
    pdr2s (3, 0);
    pclos ();

    cmov2i (100, 400);   /*  position to draw scoreboard */
```

```
    charstr ("New York 3");
    cmov2i (100, 385);
    charstr ("Boston   2");
}

/*  draw_base has not changed since the last example.                    */

draw_base (x, y)
short   x,
        y;
{
    pmv2s (x, y);
    rpdr2s (5, 0);
    rpdr2s (0, 5);
    rpdr2s (-5, 0);
    pclos ();
}
```

Workshop 3: double3.c

```
/*  double3.c  --  This program uses double buffering to draw a         */
/*                 dynamic image of a ball traveling over a field       */
/*                 without flashing.  The motion is accomplished by     */
/*                 redrawing the scene to the back buffer.  When an     */
/*                 image is completed, the entire buffer is swapped     */
/*                 to the fore, and its image is displayed.             */
/*  calls introduced:  doublebuffer(), gconfig(), swapbuffers(),        */
/*                     frontbuffer()                                    */

/*  WARNING:  For machines with a minimum configuration of bitplanes,   */
/*  there may not be enough bitplanes to run all the colors in this     */
/*  and subsequent double buffered programs.  A minimum of 12           */
/*  bitplanes are required for this program.  For subsequent programs,  */
/*  a minimum of 16 bitplanes will be required.                         */

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

/*  In the window manager, double buffered programs MUST swapbuffers,   */
/*  even when idling.  The window manager waits for all active double   */
/*  double buffered programs to issue a swapbuffers() call before       */
/*  actually swapping buffers.  If a program idles without issuing      */
/*  a swapbuffers() call, all other double buffered programs will       */
/*  wait for it.                                                        */

main () {
    initialize ();
    drawimage ();
    while (TRUE) {      /*  loop forever.  swapbuffers while looping.   */
        while (!qtest())/*  wait for input                              */
            swapbuffers();
        processinput(); /*  process input                               */
    }
}

/*  To initiate double buffering, the doublebuffer() call is made.      */
/*  The display mode change to double buffering does not take place     */
/*  until the gconfig() call is made.                                   */
```

```
initialize () {

    prefsize (450, 450);
    winopen ("diamond");

    doublebuffer();      /* display mode to become double buffer */
    gconfig();           /* display mode change takes effect     */

    qdevice( REDRAW );

    mapcolor (8, 240, 240, 240);/*  make color of arcs, lines    */
    mapcolor (9, 0, 175, 0);    /*  make color of grass          */
    mapcolor (10, 240, 150, 0); /*  make color of ball           */
}

/*  Process input from the window manager                                */

processinput() {
    short val;
    int dev;

    while (qtest()) {              /* while input on queue.            */
        dev = qread(&val);         /* read input device data.          */
        switch (dev) {
        case REDRAW:               /* window manager asking for redraw.*/
            reshapeviewport();     /* This shapes the window.          */
            drawimage();           /* This redraws the image.          */
            break;
        default:
            break;
        }
    }
}

/*  drawimage() creates motion which doesn't flash.  The for loop        */
/*  changes the value of i, which in turn, changes the position of       */
/*  the ball.  Each iteration through the loop, the field is redrawn     */
/*  and the ball is redrawn in a new position.                           */

drawimage () {
    int i;

    for (i = 0; i < 300; i = i + 3) {
        color (BLACK);          /*  clear away old field                 */
        clear ();
        diamond ();             /*  draw new field                       */
        color (10);
        circfs (i, i, 3);       /*  draw ball at position (x, y)         */
        swapbuffers();
    }
        /*  At the end of the loop, the ball will stop moving.  However,*/
        /*  the two buffers will have the ball in different positions.  */
        /*  You want to be certain that the ball is in the same          */
```

Workshop 3: double3.c

```
/*  double3.c  --   This program uses double buffering to draw a          */
/*                  dynamic image of a ball traveling over a field        */
/*                  without flashing.  The motion is accomplished by      */
/*                  redrawing the scene to the back buffer.  When an      */
/*                  image is completed, the entire buffer is swapped      */
/*                  to the fore, and its image is displayed.              */
/*  calls introduced:  doublebuffer(), gconfig(), swapbuffers(),          */
/*                     frontbuffer()                                      */

/*  WARNING:  For machines with a minimum configuration of bitplanes,     */
/*  there may not be enough bitplanes to run all the colors in this       */
/*  and subsequent double buffered programs.  A minimum of 12             */
/*  bitplanes are required for this program.  For subsequent programs,    */
/*  a minimum of 16 bitplanes will be required.                           */

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

/*  In the window manager, double buffered programs MUST swapbuffers,     */
/*  even when idling.  The window manager waits for all active double     */
/*  double buffered programs to issue a swapbuffers() call before         */
/*  actually swapping buffers.  If a program idles without issuing        */
/*  a swapbuffers() call, all other double buffered programs will         */
/*  wait for it.                                                          */

main () {
    initialize ();
    drawimage ();
    while (TRUE) {      /*  loop forever.  swapbuffers while looping.     */
        while (!qtest())/*  wait for input                                */
            swapbuffers();
        processinput(); /*  process input                                 */
    }
}

/*  To initiate double buffering, the doublebuffer() call is made.        */
/*  The display mode change to double buffering does not take place       */
/*  until the gconfig() call is made.                                     */
```

```
initialize () {

    prefsize (450, 450);
    winopen ("diamond");

    doublebuffer();      /* display mode to become double buffer */
    gconfig();           /* display mode change takes effect     */

    qdevice( REDRAW );

    mapcolor (8, 240, 240, 240);/*  make color of arcs, lines    */
    mapcolor (9, 0, 175, 0);    /*  make color of grass          */
    mapcolor (10, 240, 150, 0); /*  make color of ball           */
}

/*  Process input from the window manager                                */

processinput() {
    short val;
    int dev;

    while (qtest()) {               /* while input on queue.            */
        dev = qread(&val);          /* read input device data.          */
        switch (dev) {
        case REDRAW:                /* window manager asking for redraw.*/
            reshapeviewport();      /* This shapes the window.          */
            drawimage();            /* This redraws the image.          */
            break;
        default:
            break;
        }
    }
}

/*  drawimage() creates motion which doesn't flash.  The for loop        */
/*  changes the value of i, which in turn, changes the position of       */
/*  the ball.  Each iteration through the loop, the field is redrawn     */
/*  and the ball is redrawn in a new position.                           */

drawimage () {
    int i;

    for (i = 0; i < 300; i = i + 3) {
        color (BLACK);          /*  clear away old field                 */
        clear ();
        diamond ();             /*  draw new field                       */
        color (10);
        circfs (i, i, 3);       /*  draw ball at position (x, y)         */
        swapbuffers();
    }
        /*  At the end of the loop, the ball will stop moving.  However,*/
        /*  the two buffers will have the ball in different positions.  */
        /*  You want to be certain that the ball is in the same          */
```

```
        /*  position in both buffers.  Otherwise, when at rest, the     */
        /*  program will swap between the two buffers, and the ball     */
        /*  will appear to bounce back and forth between the disparate */
        /*  positions in the two buffers.  To draw the same scene      */
        /*  into both buffers, use frontbuffer(TRUE).  Then draw        */
        /*  the scene--that draws it into both front and back buffers. */
        /*  frontbuffer(FALSE) restores the default state, where only  */
        /*  the back buffer is drawn into.                              */
    frontbuffer(TRUE);  /*  draw final resting place of ball to         */
    color(BLACK);       /*  both buffers                                */
    clear();
    diamond();
    color (10);
    circfs (i, i, 3);   /*  draw ball at position (x, y)        */
    frontbuffer(FALSE);
}

/*  diamond() has not changed since the last example.           */

diamond () {
    color (9);          /*  make color 9 current color  */
    arcfs (0, 0, 375, 0, 900);  /*  grass               */
    color (8);          /*  make color 8 current color  */
    linewidth(2);       /*  change thickness of lines   */
    arcs (0, 0, 375, 0, 900);   /*  fences              */
    arcs (0, 0, 150, 0, 900);   /*  infield             */

    move2s (0, 0);              /*  foul lines          */
    draw2s (0, 400);
    move2s (0, 0);
    draw2s (400, 0);
    linewidth(1);       /*  restore thickness of lines  */

    circfs (43, 43, 10);        /*  pitcher's mound     */

    draw_base (90, 0);          /*  first, second and third bases      */
    draw_base (90, 90);
    draw_base (0, 90);

    pmv2s (0, 0);               /*  draw home plate     */
    pdr2s (0, 3);
    pdr2s (3, 6);
    pdr2s (6, 3);
    pdr2s (3, 0);
    pclos ();

    cmov2i (100, 400);  /*  position to draw scoreboard */
    charstr ("New York 3");
    cmov2i (100, 385);
    charstr ("Boston   2");
}

/*  draw_base has not changed since the last example.                  */
```

```
draw_base (x, y)
short   x,
        y;
{
    pmv2s (x, y);
    rpdr2s (5, 0);
    rpdr2s (0, 5);
    rpdr2s (-5, 0);
    pclos ();
}
```

Workshop 4: overlay4.c

```
/*  overlay4.c  -- This program uses double buffering to draw a          */
/*                 dynamic image of a ball traveling over a field.       */
/*                 The ability to disable the bitplanes to which the     */
/*                 field is drawn allows the field to be drawn only      */
/*                 once.  The ball is drawn and erased over the field    */
/*                 without damaging the drawing of the field.            */
/*  graphics library calls introduced:  writemask()                      */

/*  WARNING:  A minimum of 16 bitplanes are required to run this         */
/*  program.                                                             */

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

/*  The main routine has not changed since the last example.             */

main () {
    initialize ();
    drawimage ();
    while (TRUE) {      /*  loop forever.  swapbuffers while looping.   */
        while (!qtest())/*  wait for input                               */
            swapbuffers();
        processinput(); /*  process input                                */
    }
}

/*  The entries in the color map are manipulated so that when    */
/*  the ball appears over a section of the field, the ball       */
/*  stays the same RGB color.  The ball will actually appear     */
/*  as colors 16, 24 and 25 (binary 10000, 11000 and 11001),     */
/*  on top of 0, 8, 9 (binary 00000, 01000, and 01001), which    */
/*  are the colors for the field, fences and background.         */
/*  When the ball (color 16) is over either field, fence or      */
/*  black background, the writemask/overlay protects the         */
/*  color values of the underlying object, in effect, adding     */
/*  16 to the color value.  When the ball moves, only the        */
/*  16 is erased.  Mapping these three colors to the same RGB    */
/*  values makes the ball appear as one color the entire trip    */
/*  of the ball.                                                 */
```

```
initialize () {
    int i;

    prefsize (450, 450);
    winopen ("diamond");

    doublebuffer();      /* display mode to become double buffer */
    gconfig();           /* display mode change takes effect     */

    qdevice( REDRAW );

    mapcolor (8, 240, 240, 240);/*  make color of arcs, lines    */
    mapcolor (9, 0, 175, 0);    /*  make color of grass          */
    mapcolor (16, 240, 150, 0);  /*  make overlay (ball)         */
    mapcolor (24, 240, 150, 0);  /*  for colors 16, 24, and 25   */
    mapcolor (25, 240, 150, 0);
}

/*  Process input from the window manager                                */

processinput() {
    short val;
    int dev;

    while (qtest()) {                /* while input on queue.          */
        dev = qread(&val);           /* read input device data.        */
        switch (dev) {
        case REDRAW:                 /* window manager asking for redraw.*/
            reshapeviewport();       /* This shapes the window.        */
            drawimage();             /* This redraws the image.        */
            break;
        default:
            break;
        }
    }
}

/*  drawimage() draws the ball moving across the field.  The field      */
/*  is only drawn ONCE.  The colors of the field utilizes bitplanes     */
/*  number 1, 2, 3 and 4.                                               */
/*  Then the writemask() call protects bitplanes 1, 2, 3, 4 from being  */
/*  overwritten any further.  The traveling ball is written to the      */
/*  fifth bitplane (color 10000 or 16 in decimal).  For each new scene, */
/*  only the ball is cleared and drawn.  The field is never drawn       */
/*  again.                                                              */

drawimage () {
    int i;

    frontbuffer(TRUE);  /*  draw the field twice, once to each buffer   */
    color(BLACK);
    clear();
    diamond();
```

```
    frontbuffer(FALSE);
    writemask(16);      /*  protect the first four bitplanes           */
    for (i = 0; i < 300; i = i + 3) {
        color (BLACK);          /*  clear away old ball                */
        clear ();
        color (16);
        circfs (i, i, 3);       /*  draw ball at position (x, y)       */
        swapbuffers();
    }
        /*  Draw the ball into both front and back buffers.    */
    frontbuffer (TRUE); /*  draw final resting place of ball to        */
    color (BLACK);      /*  both buffers                               */
    clear ();
    color (16);
    circfs (i, i, 3);   /*  draw ball at position (x, y)       */
    frontbuffer(FALSE);
    writemask(0xfff);   /*  unprotect all bitplanes            */
}

/*  diamond() has not changed since the last example.          */

diamond () {
    color (9);          /*  make color 9 current color  */
    arcfs (0, 0, 375, 0, 900);  /*  grass               */
    color (8);          /*  make color 8 current color  */
    linewidth(2);       /*  change thickness of lines   */
    arcs (0, 0, 375, 0, 900);   /*  fences              */
    arcs (0, 0, 150, 0, 900);   /*  infield             */

    move2s (0, 0);              /*  foul lines          */
    draw2s (0, 400);
    move2s (0, 0);
    draw2s (400, 0);
    linewidth(1);       /*  restore thickness of lines  */

    circfs (43, 43, 10);        /*  pitcher's mound     */

    draw_base (90, 0);          /*  first, second and third bases      */
    draw_base (90, 90);
    draw_base (0, 90);

    pmv2s (0, 0);               /*  draw home plate     */
    pdr2s (0, 3);
    pdr2s (3, 6);
    pdr2s (6, 3);
    pdr2s (3, 0);
    pclos ();

    cmov2i (100, 400);  /*  position to draw scoreboard */
    charstr ("New York 3");
    cmov2i (100, 385);
    charstr ("Boston   2");
}
```

```
/*  draw_base has not changed since the last example.                    */

draw_base (x, y)
short   x,
        y;
{
    pmv2s (x, y);
    rpdr2s (5, 0);
    rpdr2s (0, 5);
    rpdr2s (-5, 0);
    pclos ();
}
```

Workshop 5: poll5.c

```
/*  poll5.c  -- This program polls user input from the mouse.          */
/*              The motion of the ball is initiated by the user.        */
/*              When the LEFT mouse button is pressed, the motion       */
/*              begins.                                                 */
/*  graphics library calls introduced:  getbutton()                     */

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

/*  The main routine has not changed since the last example.             */

main () {
    initialize ();
    drawimage ();
    while (TRUE) {      /*  loop forever.  swapbuffers while looping.   */
        while (!qtest()) {            /*  wait for input                */
            swapbuffers();
            if (getbutton (LEFTMOUSE)) /* We check here if the left     */
                moveball();            /* mouse has been pressed.       */
        }
        processinput(); /*  process input from queue.                   */
    }
}

/*  initialize() has not changed since the last example.                */

initialize () {
    int i;

    prefsize (450, 450);
    winopen ("diamond");

    doublebuffer();      /* display mode to become double buffer */
    gconfig();           /* display mode change takes effect     */

    qdevice( REDRAW );

    mapcolor (8, 240, 240, 240);/*  make color of arcs, lines   */
    mapcolor (9, 0, 175, 0);    /*  make color of grass         */
```

```
    mapcolor (16, 240, 150, 0);  /*  make overlay (ball)        */
    mapcolor (24, 240, 150, 0);  /*  for colors 16, 24, and 25  */
    mapcolor (25, 240, 150, 0);
}

/*  processinput() is altered to poll the mouse button          */

processinput() {
    short val;
    int dev;

    while (qtest()) {              /* while input on queue.          */
        dev = qread(&val);         /* read input device data.        */
        switch (dev) {
        case REDRAW:               /* window manager asking for redraw.*/
            reshapeviewport();     /* This shapes the window.        */
            drawimage();           /* This redraws the image.        */
            break;
        default:
            break;
        }
    }
}

/*  The former drawimage() routine is now split into two        */
/*  routines.  drawimage() draws just the baseball field.       */
/*  Also see moveball() below.                                  */

drawimage () {
    frontbuffer(TRUE);  /*  draw the field twice, once to each buffer   */
    color(BLACK);
    clear();
    diamond();
    frontbuffer(FALSE);
}

/*  The second part of the former drawimage() routine is now    */
/*  moveball().  In moveball(), the ball is sent into across    */
/*  the surface of the playing field until it settles into      */
/*  a final resting place.  Call moveball() again and again     */
/*  restarts the motion sequence.                               */

moveball() {
    int i;

    writemask(16);       /*  protect the first four bitplanes           */
    for (i = 0; i < 300; i = i + 3) {
        color (BLACK);          /*  clear away old ball                 */
        clear ();
        color (16);
        circfs (i, i, 3);       /*  draw ball at position (x, y)        */
        swapbuffers();
    }
```

```
        /*  Draw the ball into both front and back buffers.      */
    frontbuffer (TRUE); /*  draw final resting place of ball to          */
    color (BLACK);      /*  both buffers                                 */
    clear ();
    color (16);
    circfs (i, i, 3);   /*  draw ball at position (x, y)          */
    frontbuffer(FALSE);
    writemask(0xfff);   /*  unprotect all bitplanes               */
}

/*  diamond() has not changed since the last example.             */

diamond () {
    color (9);          /*  make color 9 current color  */
    arcfs (0, 0, 375, 0, 900);  /*  grass               */
    color (8);          /*  make color 8 current color  */
    linewidth(2);       /*  change thickness of lines   */
    arcs (0, 0, 375, 0, 900);   /*  fences              */
    arcs (0, 0, 150, 0, 900);   /*  infield             */

    move2s (0, 0);              /*  foul lines          */
    draw2s (0, 400);
    move2s (0, 0);
    draw2s (400, 0);
    linewidth(1);       /*  restore thickness of lines  */

    circfs (43, 43, 10);        /*  pitcher's mound     */

    draw_base (90, 0);          /*  first, second and third bases       */
    draw_base (90, 90);
    draw_base (0, 90);

    pmv2s (0, 0);               /*  draw home plate     */
    pdr2s (0, 3);
    pdr2s (3, 6);
    pdr2s (6, 3);
    pdr2s (3, 0);
    pclos ();

    cmov2i (100, 400);  /*  position to draw scoreboard */
    charstr ("New York 3");
    cmov2i (100, 385);
    charstr ("Boston   2");
}
```

```
/*  draw_base has not changed since the last example.                    */

draw_base (x, y)
short   x,
        y;
{
    pmv2s (x, y);
    rpdr2s (5, 0);
    rpdr2s (0, 5);
    rpdr2s (-5, 0);
    pclos ();
}
```

Workshop 6: aim6.c

```
/*  aim6.c  --  This program polls user input from the mouse.          */
/*              The trajectory of the ball is determined from the       */
/*              lower left hand corner of the field to a spot chosen    */
/*              by the user.  The spot is chosen by pressing the        */
/*              LEFT mouse button at some position.                     */
/*  graphics library calls introduced:  getorigin(), getvaluator()      */

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

/*  the main routine has not changed since the last example.            */

main () {
    initialize ();
    drawimage ();
    while (TRUE) {      /*  loop forever.  swapbuffers while looping.   */
        while (!qtest()) {            /*  wait for input                */
            swapbuffers();
            if (getbutton (LEFTMOUSE)) /* We check here if the left     */
                moveball();           /* mouse has been pressed.        */
        }
        processinput(); /*  process input from queue.                   */
    }
}

initialize () {
    int i;

    prefsize (450, 450);
    winopen ("diamond");

    doublebuffer();     /* display mode to become double buffer */
    gconfig();          /* display mode change takes effect     */

    qdevice( REDRAW );

    mapcolor (8, 240, 240, 240);/*  make color of arcs, lines   */
    mapcolor (9, 0, 175, 0);    /*  make color of grass         */
    mapcolor (16, 240, 150, 0);  /*  make overlay (ball)        */
```

```
    mapcolor (24, 240, 150, 0);  /*  for colors 16, 24, and 25   */
    mapcolor (25, 240, 150, 0);
}

/*  processinput() has not changed since the last example.      */

processinput() {
    short val;
    int dev;

    while (qtest()) {                /* while input on queue.          */
        dev = qread(&val);           /* read input device data.        */
        switch (dev) {
        case REDRAW:                 /* window manager asking for redraw.*/
            reshapeviewport();       /* This shapes the window.        */
            drawimage();             /* This redraws the image.        */
            break;
        default:
            break;
        }
    }
}

/*  drawimage() has not changed since the last example.        */

drawimage () {
    frontbuffer(TRUE);  /*  draw the field twice, once to each buffer   */
    color(BLACK);
    clear();
    diamond();
    frontbuffer(FALSE);
}

/*  The user can now choose the location for the trajectory of the      */
/*  ball.  The (x, y) location chosen is read from the valuators        */
/*  MOUSEX and MOUSEY.  The difference between the (x, y) location      */
/*  chosen and the (screenx, screeny) of the lower left corner of       */
/*  the field (home plate) is the distance that the ball will travel    */
/*  (distx, disty).  The ball moves 50 increments towards its final     */
/*  destination.                                                        */

moveball() {
    int i;
    int screenx, screeny;       /*  position of lower left corner        */
    int distx, disty;           /*  distance ball will travel            */
    float incrx, incry;         /*  amount to move ball in 1 iteration   */
    float newx, newy;           /*  location for ball in flight          */
    int inewx, inewy;           /*  integer location for ball in flight */

    getorigin (&screenx, &screeny);
    distx = getvaluator(MOUSEX) - screenx;
    disty = getvaluator(MOUSEY) - screeny;
    incrx = (float) distx / 50.0;
```

```
    incry = (float) disty / 50.0;
    newx = 0.0;
    newy = 0.0;
    writemask(16);      /*  protect the first four bitplanes            */
    for (i = 0; i < 50; i = i + 1) {
        newx = newx + incrx;
        newy = newy + incry;
        inewx = (int) newx;
        inewy = (int) newy;
        color (BLACK);          /*  clear away old ball                 */
        clear ();
        color (16);
        circfs (inewx, inewy, 3);/*  draw ball at position (inewx, inewy)*/
        swapbuffers();
    }
        /*  Draw the ball into both front and back buffers.     */
    frontbuffer (TRUE); /*  draw final resting place of ball to         */
    color (BLACK);      /*  both buffers                                */
    clear ();
    color (16);
    circfs (inewx, inewy, 3);/*  draw ball at position (inewx, inewy)*/
    frontbuffer(FALSE);
    writemask(0xfff);   /*  unprotect all bitplanes             */
}

/*  diamond() has not changed since the last example.           */

diamond () {
    color (9);          /*  make color 9 current color  */
    arcfs (0, 0, 375, 0, 900);  /*  grass               */
    color (8);          /*  make color 8 current color  */
    linewidth(2);       /*  change thickness of lines   */
    arcs (0, 0, 375, 0, 900);   /*  fences              */
    arcs (0, 0, 150, 0, 900);   /*  infield             */

    move2s (0, 0);              /*  foul lines          */
    draw2s (0, 400);
    move2s (0, 0);
    draw2s (400, 0);
    linewidth(1);       /*  restore thickness of lines  */

    circfs (43, 43, 10);        /*  pitcher's mound     */

    draw_base (90, 0);          /*  first, second and third bases       */
    draw_base (90, 90);
    draw_base (0, 90);

    pmv2s (0, 0);               /*  draw home plate     */
    pdr2s (0, 3);
    pdr2s (3, 6);
    pdr2s (6, 3);
    pdr2s (3, 0);
    pclos ();
```

```
    cmov2i (100, 400);  /*  position to draw scoreboard */
    charstr ("New York 3");
    cmov2i (100, 385);
    charstr ("Boston   2");
}

/*  draw_base has not changed since the last example.

draw_base (x, y)
short   x,
        y;
{
    pmv2s (x, y);
    rpdr2s (5, 0);
    rpdr2s (0, 5);
    rpdr2s (-5, 0);
    pclos ();
}
```

Workshop 7: queue7.c

```
/*  queue7.c  -- This program receives all input in an event queue.      */
/*               The effect of the program is not different from the     */
/*               previous program, aim8.  The user chooses a spot on      */
/*               the field with the LEFT MOUSE button, and the ball       */
/*               is sent to it.                                           */
/*  graphics library calls introduced:  qdevice(), tie(),                 */
/*                                      qtest(), qread()                  */

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

/*  While there is nothing on the queue, consider the program idle        */
/*  and just swapbuffers.  When something gets this programs attention,  */
/*  generally a queued device, its input will be processed.               */

main () {
    initialize ();
    drawimage ();
    while (TRUE) {       /*  loop forever.  swapbuffers and process       */
        while (!qtest())/*  input while looping                           */
            swapbuffers();
        processinput();
    }
}

/*  One line of code has been added to initialize() to            */
/*  designate the LEFTMOUSE button as a queued device.  Then      */
/*  the tie() routine queues the MOUSEX and MOUSEY.  Now          */
/*  when a LEFTMOUSE device entry is put on the queue,            */
/*  corresponding MOUSEX and MOUSEY entries are also put          */
/*  on the queue.                                                 */

initialize () {
    int i;

    prefsize (450, 450);
    winopen ("diamond");

    doublebuffer();     /* display mode to become double buffer */
```

```
    gconfig();          /* display mode change takes effect     */

    qdevice(LEFTMOUSE); /* queue the LEFTMOUSE device           */
/*  queue the MOUSEX and MOUSEY devices, and tie them to        */
/*  the LEFT MOUSE                                              */
    tie (LEFTMOUSE, MOUSEX, MOUSEY);
    qdevice( REDRAW );

    mapcolor (8, 240, 240, 240);/*  make color of arcs, lines   */
    mapcolor (9, 0, 175, 0);    /*  make color of grass         */
    mapcolor (16, 240, 150, 0);  /*  make overlay (ball)        */
    mapcolor (24, 240, 150, 0);  /*  for colors 16, 24, and 25  */
    mapcolor (25, 240, 150, 0);
}

/*  Input devices can make entries in the event queue.  The     */
/*  processinput() routine now handles those events.  The       */
/*  qtest() call reads the event queue.  qtest() immediately    */
/*  returns the device number of the first entry of the queue.  */
/*  If the queue is empty, qtest() returns FALSE, and           */
/*  processinput() is exited.  qtest() does NOT remove the      */
/*  entry from the queue.                                       */
/*  Each entry into the queue has two fields:  a device name    */
/*  and an associated data value.  qread(&val) waits until      */
/*  there is an entry on the queue, then it reads that entry.   */
/*  qread(&val) returns the name of the device and also writes  */
/*  the associated data value into the variable, val.  The      */
/*  entry on the queue is REMOVED.                              */
/*  With many buttons and keys, the values 1 and 0 are          */
/*  associated with pressing down (1) and releasing (0) a key   */
/*  or button.  With the MOUSEX and MOUSEY devices, the         */
/*  associated value is the valuator location of the cursor     */
/*  on the screen.                                              */

processinput() {
    short val;
    short mx, my;
    int dev;
    int domove;

    domove = FALSE;
    while (qtest()) {
        dev = qread(&val);
        switch (dev) {
        case LEFTMOUSE:
            if (val == 1) /*  when mouse pressed, will do motion  */
                domove = TRUE;
            break;
        case MOUSEX:
            mx = val;
            break;
        case MOUSEY:
            my = val;
```

```
            break;
        case REDRAW:                    /* window manager asking for redraw.*/
            reshapeviewport();          /* This shapes the window.          */
            drawimage();                /* This redraws the image.          */
            break;
        default:
            break;
        }
    }
    if (domove)
        moveball(mx, my);
}

/*  drawimage() has not changed since the last example.          */

drawimage () {
    frontbuffer(TRUE);  /*  draw the field twice, once to each buffer   */
    color(BLACK);
    clear();
    diamond();
    frontbuffer(FALSE);
}

/*  moveball() now takes two arguments, which are the final (mx, my)    */
/*  location of the ball on the field.                                  */

moveball(mx, my)
short mx, my;
{
    int i;
    int screenx, screeny;       /*  position of lower left corner       */
    int distx, disty;           /*  distance ball will travel           */
    float incrx, incry;         /*  amount to move ball in 1 iteration  */
    float newx, newy;           /*  location for ball in flight         */
    int inewx, inewy;           /*  integer location for ball in flight */

    getorigin (&screenx, &screeny);
    distx = mx - screenx;
    disty = my - screeny;
    incrx = (float) distx / 50.0;
    incry = (float) disty / 50.0;
    newx = 0.0;
    newy = 0.0;
    writemask(16);      /*  protect the first four bitplanes            */
    for (i = 0; i < 50; i = i + 1) {
        newx = newx + incrx;
        newy = newy + incry;
        inewx = (int) newx;
        inewy = (int) newy;
        color (BLACK);          /*  clear away old ball                 */
        clear ();
        color (16);
        circfs (inewx, inewy, 3);/*  draw ball at position (inewx, inewy)*/
```

```
        swapbuffers();
    }
        /*  Draw the ball into both front and back buffers.     */
    frontbuffer (TRUE); /*  draw final resting place of ball to         */
    color (BLACK);      /*  both buffers                                */
    clear ();
    color (16);
    circfs (inewx, inewy, 3);/*  draw ball at position (inewx, inewy)*/
    frontbuffer(FALSE);
    writemask(0xfff);   /*  unprotect all bitplanes              */
}

/*  diamond() has not changed since the last example.           */

diamond () {
    color (9);          /*  make color 9 current color  */
    arcfs (0, 0, 375, 0, 900);  /*  grass               */
    color (8);          /*  make color 8 current color  */
    linewidth(2);       /*  change thickness of lines   */
    arcs (0, 0, 375, 0, 900);   /*  fences              */
    arcs (0, 0, 150, 0, 900);   /*  infield             */

    move2s (0, 0);              /*  foul lines          */
    draw2s (0, 400);
    move2s (0, 0);
    draw2s (400, 0);
    linewidth(1);       /*  restore thickness of lines  */

    setpattern(1);      /*  change polygon fill pattern */
    circfs (43, 43, 10);        /*  pitcher's mound     */
    setpattern(0);      /*  restore poly fill pattern   */

    draw_base (90, 0);          /*  first, second and third bases       */
    draw_base (90, 90);
    draw_base (0, 90);

    pmv2s (0, 0);               /*  draw home plate     */
    pdr2s (0, 3);
    pdr2s (3, 6);
    pdr2s (6, 3);
    pdr2s (3, 0);
    pclos ();

    cmov2i (100, 400);  /*  position to draw scoreboard */
    charstr ("New York 3");
    cmov2i (100, 385);
    charstr ("Boston   2");
}
```

```
/*  draw_base has not changed since the last example.                   */

draw_base (x, y)
short   x,
        y;
{
    pmv2s (x, y);
    rpdr2s (5, 0);
    rpdr2s (0, 5);
    rpdr2s (-5, 0);
    pclos ();
}
```

Workshop 8: menu8.c

```
/*  menu8.c -- This program allows the user a pop-up menu       */
/*  interface to an application program.  In this example,      */
/*  the RIGHT MOUSE button controls a popup menu.  This popup   */
/*  menu can be used to exit the program.  Intention to exit    */
/*  the program must be reconfirmed by the user responding to   */
/*  a second popup menu.  Popup menu support is ONLY available  */
/*  in the window manager.                                      */
/*  graphics library calls introduced:  defpup(), newpup(),     */
/*              addtopup(), dopup().                            */

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

/*  new global variables for menus              */

int     menu,
        killmenu;

/*  the main() routine has not changed since the last example.  */

main () {
    initialize ();
    drawimage ();
    while (TRUE) {      /*  loop forever.  swapbuffers and process      */
        while (!qtest())/*  input while looping                         */
            swapbuffers();
        processinput();
    }
}

/*  In initialize(), queue the RIGHTMOUSE button for popup menus.       */
/*  Also the pup calls are used to create popup menus.                  */

initialize () {
    int i;

    prefsize (450, 450);
    winopen ("diamond");
```

```
    doublebuffer();      /* display mode to become double buffer */
    gconfig();           /* display mode change takes effect     */

    qdevice(LEFTMOUSE); /* queue the LEFTMOUSE device           */
    qdevice(RIGHTMOUSE);/* queue the RIGHTMOUSE device          */
/*  queue the MOUSEX and MOUSEY devices, and tie them to        */
/*  the LEFT MOUSE                                              */
    tie (LEFTMOUSE, MOUSEX, MOUSEY);
    qdevice(REDRAW);     /* queue the REDRAW device             */

    mapcolor (8, 240, 240, 240);/*  make color of arcs, lines   */
    mapcolor (9, 0, 175, 0);    /*  make color of grass         */
    mapcolor (16, 240, 150, 0);  /*  make overlay (ball)        */
    mapcolor (24, 240, 150, 0);  /*  for colors 16, 24, and 25  */
    mapcolor (25, 240, 150, 0);

/*  Now we define two pop-up menus using two different methods. The 1st */
/*  method defines the menu and the text describing the options, all in */
/*  the same step, using a defpup () call. The 2nd method first defines */
/*  the menu using a newpup () call, then puts in the text using a      */
/*  addtopup () all.                                                    */

/*  Note: the '|' character separates menu items, and the %t flag is    */
/*  used to designate the menu title.                                   */

/*  newpup () and defpup () both return the menu id number used later.  */

    killmenu = defpup ("Do you want to exit? %t|yes|no");
    menu = newpup ();
    addtopup (menu, "Baseball %t|exit program");

}

/*  In processinput(), the RIGHTMOUSE button, when it is pressed        */
/*  down (val == 1), displays a popup menu.  The value returned         */
/*  by the user's selection is stored in variable menuval.  The         */
/*  value returned may be:                                              */
/*       1 -- for exit program                                          */
/*      -1 -- nothing on menu is chosen                                 */
/*  Then the user is asked to reconfirm an exit program decision.       */
/*       1 -- for yes                                                   */
/*       2 -- for no                                                    */
/*      -1 -- nothing on menu is chosen                                 */

processinput() {
    short val;
    short mx, my;
    int dev;
    int domove;
    int menuval, kmenuval;

    domove = FALSE;
    while (qtest()) {
```

```
        dev = qread(&val);
        switch (dev) {
        case LEFTMOUSE:
            if (val == 1) /*  when mouse pressed, will do motion  */
                domove = TRUE;
            break;
        case MOUSEX:
            mx = val;
            break;
        case MOUSEY:
            my = val;
            break;
        case RIGHTMOUSE:  /*  when RIGHT MOUSE pressed, do popup  */
            if (val == 1) {
                menuval = dopup(menu); /* returns the menu item chosen */
                if (menuval == 1) {  /*  if exit chosen, ask user */
                    kmenuval = dopup (killmenu); /*  to reconfirm */
                    if (kmenuval == 1)
                        exit (0);
                }
            }
            break;
        case REDRAW:
            reshapeviewport();
            drawimage();
            break;
        default:
            break;
        }
    }
    if (domove)
        moveball(mx, my);
}

/*  drawimage() has not changed since the last example.         */

drawimage () {
    frontbuffer(TRUE);  /*  draw the field twice, once to each buffer   */
    color(BLACK);
    clear();
    diamond();
    frontbuffer(FALSE);
}

/*  moveball() has not changed since the last example.          */

moveball(mx, my)
short mx, my;
{
    int i;
    int screenx, screeny;       /*  position of lower left corner        */
    int distx, disty;           /*  distance ball will travel            */
    float incrx, incry;         /*  amount to move ball in 1 iteration   */
```

```
    float newx, newy;           /*  location for ball in flight         */
    int inewx, inewy;           /*  integer location for ball in flight */

    getorigin (&screenx, &screeny);
    distx = mx - screenx;
    disty = my - screeny;
    incrx = (float) distx / 50.0;
    incry = (float) disty / 50.0;
    newx = 0.0;
    newy = 0.0;
    writemask(16);      /*  protect the first four bitplanes            */
    for (i = 0; i < 50; i = i + 1) {
        newx = newx + incrx;
        newy = newy + incry;
        inewx = (int) newx;
        inewy = (int) newy;
        color (BLACK);          /*  clear away old ball                 */
        clear ();
        color (16);
        circfs (inewx, inewy, 3);/*  draw ball at position (inewx, inewy)*/
        swapbuffers();
    }
        /*  Draw the ball into both front and back buffers.     */
    frontbuffer (TRUE); /*  draw final resting place of ball to         */
    color (BLACK);      /*  both buffers                                */
    clear ();
    color (16);
    circfs (inewx, inewy, 3);/*  draw ball at position (inewx, inewy)*/
    frontbuffer(FALSE);
    writemask(0xfff);   /*  unprotect all bitplanes             */
}

/*  diamond() has not changed since the last example.          */

diamond () {
    color (9);          /*  make color 9 current color  */
    arcfs (0, 0, 375, 0, 900);  /*  grass               */
    color (8);          /*  make color 8 current color  */
    linewidth(2);       /*  change thickness of lines   */
    arcs (0, 0, 375, 0, 900);   /*  fences              */
    arcs (0, 0, 150, 0, 900);   /*  infield             */

    move2s (0, 0);              /*  foul lines          */
    draw2s (0, 400);
    move2s (0, 0);
    draw2s (400, 0);
    linewidth(1);       /*  restore thickness of lines  */

    circfs (43, 43, 10);        /*  pitcher's mound     */

    draw_base (90, 0);          /*  first, second and third bases       */
    draw_base (90, 90);
    draw_base (0, 90);
```

```
    pmv2s (0, 0);               /*  draw home plate     */
    pdr2s (0, 3);
    pdr2s (3, 6);
    pdr2s (6, 3);
    pdr2s (3, 0);
    pclos ();

    cmov2i (100, 400);  /*  position to draw scoreboard */
    charstr ("New York 3");
    cmov2i (100, 385);
    charstr ("Boston   2");
}

/*  draw_base has not changed since the last example.                   */

draw_base (x, y)
short   x,
        y;
{
    pmv2s (x, y);
    rpdr2s (5, 0);
    rpdr2s (0, 5);
    rpdr2s (-5, 0);
    pclos ();
}
```

Workshop 9: threed9.c

```
/*  threed9.c -- This is a version of the previous program      */
/*  using 3D floating point calls instead of 2D integer         */
/*  graphics commands.  We did NOT have to do this at all       */
/*  since the z value for 2D graphics commands defaults to z=0. */
/*  However, it is very likely that you wil' be building        */
/*  objects in 3D space with floating point, rather than        */
/*  integer coordinates.  This change was made to introduce     */
/*  you to 3D calls.                                            */
/*  graphics library calls introduced:  arc, arcf, circf, cmov, */
/*      draw, move, pdr, pmv, rpdr.  All these are 3D floating  */
/*      point versions of their 2D integer relatives.           */

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

/*  global variables for menus          */

int     menu,
        killmenu;

/*  the main() routine has not changed since the last example.  */

main () {
    initialize ();
    drawimage ();
    while (TRUE) {      /*  loop forever.  swapbuffers and process      */
        while (!qtest())/*  input while looping                         */
            swapbuffers();
        processinput();
    }
}

/*  initialize() has not changed since the last example.        */

initialize () {
    int i;

    prefsize (450, 450);
    winopen ("diamond");
```

```
    doublebuffer();     /* display mode to become double buffer */
    gconfig();          /* display mode change takes effect     */

    qdevice(LEFTMOUSE); /* queue the LEFTMOUSE device           */
    qdevice(RIGHTMOUSE);/* queue the RIGHTMOUSE device          */
/*  queue the MOUSEX and MOUSEY devices, and tie them to        */
/*  the LEFT MOUSE                                              */
    tie (LEFTMOUSE, MOUSEX, MOUSEY);
    qdevice(REDRAW);    /* queue the REDRAW device              */

    mapcolor (8, 240, 240, 240);/*  make color of arcs, lines   */
    mapcolor (9, 0, 175, 0);    /*  make color of grass         */
    mapcolor (16, 240, 150, 0);  /*  make overlay (ball)        */
    mapcolor (24, 240, 150, 0);  /*  for colors 16, 24, and 25  */
    mapcolor (25, 240, 150, 0);

    killmenu = defpup ("Do you want to exit? %t|yes|no");
    menu = newpup ();
    addtopup (menu, "Baseball %t|exit program");
}

/*  processinput() has not changed since the last example.      */

processinput() {
    short val;
    short mx, my;
    int dev;
    int domove;
    int menuval, kmenuval;

    domove = FALSE;
    while (qtest()) {
        dev = qread(&val);
        switch (dev) {
        case LEFTMOUSE:
            if (val == 1) /*  when mouse pressed, will do motion  */
                domove = TRUE;
            break;
        case MOUSEX:
            mx = val;
            break;
        case MOUSEY:
            my = val;
            break;
        case RIGHTMOUSE:
            if (val == 1) {
                menuval = dopup(menu);
                if (menuval == 1) {  /*  if exit chosen, ask user */
                    kmenuval = dopup (killmenu); /*  to reconfirm */
                    if (kmenuval == 1)
                        exit (0);
                }
            }
```

```
            break;
        case REDRAW:
            reshapeviewport();
            drawimage();
            break;
        default:
            break;
        }
    }
    if (domove)
        moveball(mx, my);
}

/*  drawimage() has not changed since the last example.         */

drawimage () {
    frontbuffer(TRUE);  /*  draw the field twice, once to each buffer   */
    color(BLACK);
    clear();
    diamond();
    frontbuffer(FALSE);
}

/*  moveball() has changed to 3D floating point         */

moveball(mx, my)
short mx, my;
{
    int i;
    int screenx, screeny;       /*  position of lower left corner         */
    int distx, disty;           /*  distance ball will travel             */
    float incrx, incry;         /*  amount to move ball in 1 iteration    */
    float newx, newy;           /*  location for ball in flight           */

    getorigin (&screenx, &screeny);
    distx = mx - screenx;
    disty = my - screeny;
    incrx = (float) distx / 50.0;
    incry = (float) disty / 50.0;
    newx = 0.0;
    newy = 0.0;
    writemask(16);      /*  protect the first four bitplanes              */
    for (i = 0; i < 50; i = i + 1) {
        newx = newx + incrx;
        newy = newy + incry;
        color (BLACK);          /*  clear away old ball                   */
        clear ();
        color (16);
        circf (newx, newy, 3.0);/*  draw ball at position (newx, newy)    */
        swapbuffers();
    }
        /*  Draw the ball into both front and back buffers.     */
    frontbuffer (TRUE); /*  draw final resting place of ball to         */
```

```
    color (BLACK);      /*  both buffers                                    */
    clear ();
    color (16);
    circf (newx, newy, 3.0);/*  draw ball at position (newx, newy)         */
    frontbuffer(FALSE);
    writemask(0xfff);   /*  unprotect all bitplanes                  */
}

/*  diamond() has changed to 3D floating point  */

diamond () {
    color (9);          /*  make color 9 current color  */
    arcf (0.0, 0.0, 375.0, 0, 900);     /*  grass       */
    color (8);          /*  make color 8 current color  */
    linewidth(2);       /*  change thickness of lines   */
    arc (0.0, 0.0, 375.0, 0, 900);      /*  fences      */
    arc (0.0, 0.0, 150.0, 0, 900);      /*  infield     */

    move (0.0, 0.0, 0.0);       /*  foul lines          */
    draw (0.0, 400.0, 0.0);
    move (0.0, 0.0, 0.0);
    draw (400.0, 0.0, 0.0);
    linewidth(1);       /*  restore thickness of lines  */

    circf (43.0, 43.0, 10.0);   /*  pitcher's mound     */

    draw_base (90.0, 0.0, 0.0); /*  first, second and third bases       */
    draw_base (90.0, 90.0, 0.0);
    draw_base (0.0, 90.0, 0.0);

    pmv (0.0, 0.0, 0.0);        /*  draw home plate     */
    pdr (0.0, 3.0, 0.0);
    pdr (3.0, 6.0, 0.0);
    pdr (6.0, 3.0, 0.0);
    pdr (3.0, 0.0, 0.0);
    pclos ();

    cmov (100.0, 400.0, 0.0);   /*  position to draw scoreboard */
    charstr ("New York 3");
    cmov (100.0, 385.0, 0.0);
    charstr ("Boston   2");
}
```

```
/*  draw_base() has changed to 3D floating point         */

draw_base (x, y, z)
float   x,
        y,
        z;
{
    pmv (x, y, z);
    rpdr (5.0, 0.0, 0.0);
    rpdr (0.0, 5.0, 0.0);
    rpdr (-5.0, 0.0, 0.0);
    pclos ();
}
```

Workshop 10: coord10.c

```
/*  project10.c -- This program introduces projection          */
/*  transformations.  The ortho() call in drawimage() fits     */
/*  the 425.0 x 425.0 field into the opened window. The        */
/*  field will shrink or stretch to fit the window. This       */
/*  makes the prefsize() call unnecessary.                     */
/*  Some adjustments to the moveball() routine were made       */
/*  so that the mouse position corresponds to the location     */
/*  on the field.                                              */
/*  graphics library calls introduced:  ortho(), keepaspect()  */

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

/*  global variables for menus          */

int     menu,
        killmenu;

/*  the main() routine has not changed since the last example.  */

main () {
    initialize ();
    drawimage ();
    while (TRUE) {      /*  loop forever.  swapbuffers and process      */
        while (!qtest())/*  input while looping                         */
            swapbuffers();
        processinput();
    }
}

/*  Using the ortho() call has made the prefsize() call unnecessary      */
/*  so it has been removed.  The keepaspect() call fixes the shape       */
/*  of the opened window to a square.                                    */

initialize () {
    int i;

    keepaspect (1, 1);
    winopen ("diamond");
```

```
    doublebuffer();      /* display mode to become double buffer */
    gconfig();           /* display mode change takes effect     */

    qdevice(LEFTMOUSE); /* queue the LEFTMOUSE device            */
    qdevice(RIGHTMOUSE);/* queue the RIGHTMOUSE device           */
/*  queue the MOUSEX and MOUSEY devices, and tie them to        */
/*  the LEFT MOUSE                                              */
    tie (LEFTMOUSE, MOUSEX, MOUSEY);
    qdevice(REDRAW);     /* queue the REDRAW device              */

    mapcolor (8, 240, 240, 240);/*  make color of arcs, lines   */
    mapcolor (9, 0, 175, 0);     /*  make color of grass         */
    mapcolor (16, 240, 150, 0);  /*  make overlay (ball)         */
    mapcolor (24, 240, 150, 0);  /*  for colors 16, 24, and 25   */
    mapcolor (25, 240, 150, 0);

    killmenu = defpup ("Do you want to exit? %t|yes|no");
    menu = newpup ();
    addtopup (menu, "Baseball %t|exit program");
}

/*  processinput() has not changed since the last example.      */

processinput() {
    short val;
    short mx, my;
    int dev;
    int domove;
    int menuval, kmenuval;

    domove = FALSE;
    while (qtest()) {
        dev = qread(&val);
        switch (dev) {
        case LEFTMOUSE:
            if (val == 1) /*  when mouse pressed, will do motion  */
                domove = TRUE;
            break;
        case MOUSEX:
            mx = val;
            break;
        case MOUSEY:
            my = val;
            break;
        case RIGHTMOUSE:
            if (val == 1) {
                menuval = dopup(menu);
                if (menuval == 1) {  /*  if exit chosen, ask user */
                    kmenuval = dopup (killmenu); /*  to reconfirm */
                    if (kmenuval == 1)
                        exit (0);
                }
            }
```

```
            break;
        case REDRAW:
            reshapeviewport();
            drawimage();
            break;
        default:
            break;
        }
    }
    if (domove)
        moveball(mx, my);
}

/*  An ortho() command has been added to drawimage().  The left,         */
/*  right, top and bottom parameters for ortho specify what the          */
/*  section of the world is displayed in the opened window.  The         */
/*  z values are set such that the ball doesn't get clipped no matter    */
/*  how close or far away the ball is from us. Specifically the z        */
/*  clipping planes have been set to -10000.0 units behind us and        */
/*  10000.0 units in front of us.                                        */

drawimage () {
    ortho (0.0, 425.0, 0.0, 425.0, -10000.0, 10000.0);
    frontbuffer(TRUE);  /*  draw the field twice, once to each buffer    */
    color(BLACK);
    clear();
    diamond();
    frontbuffer(FALSE);
}

/*  moveball() is substantially different from the last example.         */
/*  We can no longer pick an absolute (x, y) location on the screen      */
/*  with the mouse.  We can pick a location on the window and            */
/*  calculate the relative distance travelled, depending on the size     */
/*  of the window.  Note that rdistx and rdisty are the relative         */
/*  distance:  the absolute distance * 425.0 (world space size of        */
/*  field) / screen size of window                                       */

moveball(mx, my)
short mx, my;
{
    int i;
    int screenx, screeny;       /*  position of lower left corner        */
    int sizex, sizey;           /*  size of window                       */
    int distx, disty;           /*  distance ball will travel            */
    float rdistx, rdisty;       /*  relative distance ball will travel   */
    float incrx, incry;         /*  amount to move ball in 1 iteration   */
    float newx, newy;           /*  location for ball in flight          */

    getorigin (&screenx, &screeny); /* coordinates of lower left corner */
    getsize (&sizex, &sizey);       /* size of window                   */
    distx = mx - screenx;           /* distance to x mouse position     */
    disty = my - screeny;           /* distance to y mouse position     */
```

```
    /* This stuff calculates the relative position of the ball in the    */
    /* window as defined by the window size and projection.               */
    rdistx = (float) distx * 425.0 / (float) sizex;
    rdisty = (float) disty * 425.0 / (float) sizey;

    incrx = (float) rdistx / 50.0; /*  do 50 iterations          */
    incry = (float) rdisty / 50.0;
    newx = 0.0;
    newy = 0.0;
    writemask(16);      /*  protect the first four bitplanes          */
    for (i = 0; i < 50; i = i + 1) {
        newx = newx + incrx;
        newy = newy + incry;
        color (BLACK);          /*  clear away old ball               */
        clear ();
        color (16);
        circf (newx, newy, 3.0);/*  draw ball at position (newx, newy)  */
        swapbuffers();
    }
        /*  Draw the ball into both front and back buffers.    */
    frontbuffer (TRUE); /*  draw final resting place of ball to       */
    color (BLACK);      /*  both buffers                              */
    clear ();
    color (16);
    circf (newx, newy, 3.0);/*  draw ball at position (newx, newy)     */
    frontbuffer(FALSE);
    writemask(0xfff);   /*  unprotect all bitplanes          */
}

/*  diamond() has not changed since the last example.          */

diamond () {
    color (9);          /*  make color 9 current color  */
    arcf (0.0, 0.0, 375.0, 0, 900);     /*  grass       */
    color (8);          /*  make color 8 current color  */
    linewidth(2);       /*  change thickness of lines   */
    arc (0.0, 0.0, 375.0, 0, 900);      /*  fences      */
    arc (0.0, 0.0, 150.0, 0, 900);      /*  infield     */

    move (0.0, 0.0, 0.0);       /*  foul lines          */
    draw (0.0, 400.0, 0.0);
    move (0.0, 0.0, 0.0);
    draw (400.0, 0.0, 0.0);
    linewidth(1);       /*  restore thickness of lines  */

    circf (43.0, 43.0, 10.0);   /*  pitcher's mound     */

    draw_base (90.0, 0.0, 0.0); /*  first, second and third bases        */
    draw_base (90.0, 90.0, 0.0);
    draw_base (0.0, 90.0, 0.0);
```

```
    pdr (0.0, 3.0, 0.0);
    pdr (3.0, 6.0, 0.0);
    pdr (6.0, 3.0, 0.0);
    pdr (3.0, 0.0, 0.0);
    pclos ();

    cmov (100.0, 400.0, 0.0);   /*  position to draw scoreboard */
    charstr ("New York 3");
    cmov (100.0, 385.0, 0.0);
    charstr ("Boston   2");
}

/*  draw_base() has not changed since the last example.         */

draw_base (x, y, z)
float   x,
        y,
        z;
{
    pmv (x, y, z);
    rpdr (5.0, 0.0, 0.0);
    rpdr (0.0, 5.0, 0.0);
    rpdr (-5.0, 0.0, 0.0);
    pclos ();
}
```

Workshop 11: translate11.c

```
/*  translate11.c -- translate11 demonstrates and introduces     */
/*  modeling transformations.  The translate command is being    */
/*  used to move the ball into different positions along its     */
/*  flight path.  For each frame, the ball is translated from    */
/*  the origin to its position in space.                         */
/*  moveball() is the only routine changed.                      */
/*  graphics commands introduced:  translate()                   */

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

/*  global variables for menus          */

int     menu,
        killmenu;

/*  the main() routine has not changed since the last example.  */

main () {
    initialize ();
    drawimage ();
    while (TRUE) {      /*  loop forever.  swapbuffers and process      */
        while (!qtest())/*  input while looping                         */
            swapbuffers();
        processinput();
    }
}

/*  the initialize() routine has not changed since the last example.    */

initialize () {
    int i;

    keepaspect (1, 1);
    winopen ("diamond");

    doublebuffer();     /* display mode to become double buffer */
    gconfig();          /* display mode change takes effect     */
```

```
    qdevice(LEFTMOUSE); /* queue the LEFTMOUSE device            */
    qdevice(RIGHTMOUSE);/* queue the RIGHTMOUSE device           */
/*  queue the MOUSEX and MOUSEY devices, and tie them to         */
/*  the LEFT MOUSE                                               */
    tie (LEFTMOUSE, MOUSEX, MOUSEY);
    qdevice(REDRAW);    /* queue the REDRAW device               */

    mapcolor (8, 240, 240, 240);/*  make color of arcs, lines    */
    mapcolor (9, 0, 175, 0);    /*  make color of grass          */
    mapcolor (16, 240, 150, 0);  /*  make overlay (ball)         */
    mapcolor (24, 240, 150, 0);  /*  for colors 16, 24, and 25   */
    mapcolor (25, 240, 150, 0);

    killmenu = defpup ("Do you want to exit? %t|yes|no");
    menu = newpup ();
    addtopup (menu, "Baseball %t|exit program");
}

/*  processinput() has not changed since the last example.       */

processinput() {
    short val;
    short mx, my;
    int dev;
    int domove;
    int menuval, kmenuval;

    domove = FALSE;
    while (qtest()) {
        dev = qread(&val);
        switch (dev) {
        case LEFTMOUSE:
            if (val == 1) /*  when mouse pressed, will do motion  */
                domove = TRUE;
            break;
        case MOUSEX:
            mx = val;
            break;
        case MOUSEY:
            my = val;
            break;
        case RIGHTMOUSE:
            if (val == 1) {
                menuval = dopup(menu);
                if (menuval == 1) {  /*  if exit chosen, ask user */
                    kmenuval = dopup (killmenu); /*  to reconfirm */
                    if (kmenuval == 1)
                        exit (0);
                }
            }
            break;
        case REDRAW:
            reshapeviewport();
```

```
                drawimage();
                break;
            default:
                break;
            }
        }
        if (domove)
            moveball(mx, my);
}

/*  the drawimage() routine has not changed since the last example.     */

drawimage () {
    ortho (0.0, 425.0, 0.0, 425.0, -10000.0, 10000.0);
    frontbuffer(TRUE);  /*  draw the field twice, once to each buffer   */
    color(BLACK);
    clear();
    diamond();
    frontbuffer(FALSE);
}

/*  The translate() command is now used moveball() to position the      */
/*  The ball is a circle, which would be drawn at the origin, but is    */
/*  then translated to different positions along its flight path.       */

moveball(mx, my)
short mx, my;
{
    int i;
    int screenx, screeny;       /*  position of lower left corner       */
    int sizex, sizey;           /*  size of window                      */
    int distx, disty;           /*  distance ball will travel           */
    float rdistx, rdisty;       /*  relative distance ball will travel  */
    float incrx, incry;         /*  amount to move ball in 1 iteration  */
    float newx, newy;           /*  location for ball in flight         */

    getorigin (&screenx, &screeny); /* coordinates of lower left corner */
    getsize (&sizex, &sizey);       /* size of window                   */
    distx = mx - screenx;           /* distance to x mouse position     */
    disty = my - screeny;           /* distance to y mouse position     */

    /* This stuff calculates the relative position of the ball in the   */
    /* window as defined by the window size and projection.             */
    rdistx = (float) distx * 425.0 / (float) sizex;
    rdisty = (float) disty * 425.0 / (float) sizey;

    incrx = (float) rdistx / 50.0; /*  do 50 iterations         */
    incry = (float) rdisty / 50.0;
    newx = 0.0;
    newy = 0.0;
    writemask(16);      /*  protect the first four bitplanes            */
    for (i = 0; i < 50; i = i + 1) {
        newx = newx + incrx;
```

```
        newy = newy + incry;
        color (BLACK);          /*  clear away old ball                  */
        clear ();
        color (16);
        pushmatrix ();
        translate (newx, newy, 0.0);/*  translate local axes to newx, newy*/
        circf (0.0, 0.0, 3.0);/*  draw ball at origin (0.0, 0.0)         */
        popmatrix ();
        swapbuffers();
    }
        /*  Draw the ball into both front and back buffers.    */
    frontbuffer (TRUE); /*  draw final resting place of ball to          */
    color (BLACK);      /*  both buffers                                 */
    clear ();
    color (16);
    pushmatrix ();
    translate (newx, newy, 0.0);/*  translate local axes to newx, newy  */
    circf (0.0, 0.0, 3.0);/*  draw ball at origin (0.0, 0.0)    */
    popmatrix ();
    frontbuffer(FALSE);
    writemask(0xfff);   /*  unprotect all bitplanes             */
}

/*  diamond() has not changed since the last example.          */

diamond () {
    color (9);          /*  make color 9 current color  */
    arcf (0.0, 0.0, 375.0, 0, 900);     /*  grass       */
    color (8);          /*  make color 8 current color  */
    linewidth(2);       /*  change thickness of lines   */
    arc (0.0, 0.0, 375.0, 0, 900);      /*  fences      */
    arc (0.0, 0.0, 150.0, 0, 900);      /*  infield     */

    move (0.0, 0.0, 0.0);       /*  foul lines          */
    draw (0.0, 400.0, 0.0);
    move (0.0, 0.0, 0.0);
    draw (400.0, 0.0, 0.0);
    linewidth(1);       /*  restore thickness of lines  */

    circf (43.0, 43.0, 10.0);   /*  pitcher's mound     */

    draw_base (90.0, 0.0, 0.0); /*  first, second and third bases       */
    draw_base (90.0, 90.0, 0.0);
    draw_base (0.0, 90.0, 0.0);

    pmv (0.0, 0.0, 0.0);        /*  draw home plate     */
    pdr (0.0, 3.0, 0.0);
    pdr (3.0, 6.0, 0.0);
    pdr (6.0, 3.0, 0.0);
    pdr (3.0, 0.0, 0.0);
    pclos ();
```

```
    cmov (100.0, 400.0, 0.0);   /*  position to draw scoreboard */
    charstr ("New York 3");
    cmov (100.0, 385.0, 0.0);
    charstr ("Boston   2");
}

/*  draw_base() has not changed since the last example.        */

draw_base (x, y, z)
float   x,
        y,
        z;
{
    pmv (x, y, z);
    rpdr (5.0, 0.0, 0.0);
    rpdr (0.0, 5.0, 0.0);
    rpdr (-5.0, 0.0, 0.0);
    pclos ();
}
```

Workshop 12: composite12.c

```
/*  composite12.c -- composite12 has a different looking ball   */
/*  than was found in previous workshops.  It demonstrates      */
/*  compositing modeling transformations.  In this case,        */
/*  translates and rotates are composited.                      */
/*  The translate command moves the ball into different         */
/*  positions along its flight path.  Remember that we can      */
/*  say the translate command moves the coordinate system.      */
/*  Then circles are drawn at that new location.  Rotate        */
/*  commands are composited to the current matrix to draw       */
/*  each new circle in a new position, although still centered  */
/*  around the moved coordinate system.                         */
/*  The program works like translate11, but the ball appears    */
/*  different.                                                  */
/*  move_ball() is the only changed procedure.  draw_ball()     */
/*  is a new routine for drawing the ball at the origin.        */
/*  graphics commands introduced:  rotate()                     */

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

/*  global variables for menus          */

int     menu,
        killmenu;

/*  the main() routine has not changed since the last example.  */

main () {
    initialize ();
    drawimage ();
    while (TRUE) {      /*  loop forever.  swapbuffers and process      */
        while (!qtest())/*  input while looping                         */
            swapbuffers();
        processinput();
    }
}

/*  the initialize() routine has not changed since the last example.    */
```

```
initialize () {
    int i;

    keepaspect (1, 1);
    winopen ("diamond");

    doublebuffer();      /* display mode to become double buffer */
    gconfig();           /* display mode change takes effect     */

    qdevice(LEFTMOUSE); /* queue the LEFTMOUSE device            */
    qdevice(RIGHTMOUSE);/* queue the RIGHTMOUSE device           */
/*  queue the MOUSEX and MOUSEY devices, and tie them to         */
/*  the LEFT MOUSE                                               */
    tie (LEFTMOUSE, MOUSEX, MOUSEY);
    qdevice(REDRAW);     /* queue the REDRAW device              */

    mapcolor (8, 240, 240, 240);/*  make color of arcs, lines   */
    mapcolor (9, 0, 175, 0);    /*  make color of grass          */
    mapcolor (16, 240, 150, 0);  /*  make overlay (ball)         */
    mapcolor (24, 240, 150, 0);  /*  for colors 16, 24, and 25   */
    mapcolor (25, 240, 150, 0);

    killmenu = defpup ("Do you want to exit? %t|yes|no");
    menu = newpup ();
    addtopup (menu, "Baseball %t|exit program");
}

/*  processinput() has not changed since the last example.       */

processinput() {
    short val;
    short mx, my;
    int dev;
    int domove;
    int menuval, kmenuval;

    domove = FALSE;
    while (qtest()) {
        dev = qread(&val);
        switch (dev) {
        case LEFTMOUSE:
            if (val == 1) /*  when mouse pressed, will do motion  */
                domove = TRUE;
            break;
        case MOUSEX:
            mx = val;
            break;
        case MOUSEY:
            my = val;
            break;
        case RIGHTMOUSE:
            if (val == 1) {
                menuval = dopup(menu);
```

```
                if (menuval == 1) {  /*  if exit chosen, ask user */
                    kmenuval = dopup (killmenu); /*  to reconfirm */
                    if (kmenuval == 1)
                        exit (0);
                }
            }
            break;
        case REDRAW:
            reshapeviewport();
            drawimage();
            break;
        default:
            break;
        }
    }
    if (domove)
        moveball(mx, my);
}

/*  the drawimage() routine has not changed since the last example.     */

drawimage () {
    ortho (0.0, 425.0, 0.0, 425.0, -10000.0, 10000.0);
    frontbuffer(TRUE);  /*  draw the field twice, once to each buffer   */
    color(BLACK);
    clear();
    diamond();
    frontbuffer(FALSE);
}

/*  The moveball() routine no longer has a circf (0.0, 0.0, 3.0) to     */
/*  draw the ball.  Instead, the draw_ball() routine is called to       */
/*  draw the ball at the origin.                                        */

moveball(mx, my)
short mx, my;
{
    int i;
    int screenx, screeny;       /*  position of lower left corner       */
    int sizex, sizey;           /*  size of window                      */
    int distx, disty;           /*  distance ball will travel           */
    float rdistx, rdisty;       /*  relative distance ball will travel  */
    float incrx, incry;         /*  amount to move ball in 1 iteration  */
    float newx, newy;           /*  location for ball in flight         */

    getorigin (&screenx, &screeny); /* coordinates of lower left corner */
    getsize (&sizex, &sizey);       /* size of window                   */
    distx = mx - screenx;           /* distance to x mouse position     */
    disty = my - screeny;           /* distance to y mouse position     */

    /* This stuff calculates the relative position of the ball in the   */
    /* window as defined by the window size and projection.             */
    rdistx = (float) distx * 425.0 / (float) sizex;
```

```
    rdisty = (float) disty * 425.0 / (float) sizey;

    incrx = (float) rdistx / 50.0; /*  do 50 iterations         */
    incry = (float) rdisty / 50.0;
    newx = 0.0;
    newy = 0.0;
    writemask(16);       /*  protect the first four bitplanes          */
    for (i = 0; i < 50; i = i + 1) {
        newx = newx + incrx;
        newy = newy + incry;
        color (BLACK);          /*  clear away old ball                */
        clear ();
        color (16);
        pushmatrix ();
        translate (newx, newy, 0.0);/*  translate local axes to newx, newy*/
        draw_ball ();
        popmatrix ();
        swapbuffers();
    }
        /*  Draw the ball into both front and back buffers.      */
    frontbuffer (TRUE); /*  draw final resting place of ball to        */
    color (BLACK);      /*  both buffers                               */
    clear ();
    color (16);
    pushmatrix ();
    translate (newx, newy, 0.0);/*  translate local axes to newx, newy  */
    draw_ball ();
    popmatrix ();
    frontbuffer(FALSE);
    writemask(0xfff);   /*  unprotect all bitplanes              */
}

/*  draw_ball() is a new routine.  The ball is drawn at the     */
/*  origin (home plate), and translated into position.  At      */
/*  that translated position, four circles are put down.  The   */
/*  four circles are actually the same circle, each slightly    */
/*  rotated from the previously drawn circle.                   */

draw_ball ()
{
    circ (0.0, 0.0, 3.0);
    rotate (900, 'y');
    circ (0.0, 0.0, 3.0);
    rotate (600, 'x');
    circ (0.0, 0.0, 3.0);
    rotate (600, 'x');
    circ (0.0, 0.0, 3.0);
}
```

```
/*  diamond() has not changed since the last example.                */

diamond () {
    color (9);          /*  make color 9 current color  */
    arcf (0.0, 0.0, 375.0, 0, 900);     /*  grass        */
    color (8);          /*  make color 8 current color  */
    linewidth(2);       /*  change thickness of lines   */
    arc (0.0, 0.0, 375.0, 0, 900);      /*  fences       */
    arc (0.0, 0.0, 150.0, 0, 900);      /*  infield      */

    move (0.0, 0.0, 0.0);       /*  foul lines          */
    draw (0.0, 400.0, 0.0);
    move (0.0, 0.0, 0.0);
    draw (400.0, 0.0, 0.0);
    linewidth(1);       /*  restore thickness of lines  */

    circf (43.0, 43.0, 10.0);   /*  pitcher's mound     */

    draw_base (90.0, 0.0, 0.0); /*  first, second and third bases       */
    draw_base (90.0, 90.0, 0.0);
    draw_base (0.0, 90.0, 0.0);

    pmv (0.0, 0.0, 0.0);        /*  draw home plate     */
    pdr (0.0, 3.0, 0.0);
    pdr (3.0, 6.0, 0.0);
    pdr (6.0, 3.0, 0.0);
    pdr (3.0, 0.0, 0.0);
    pclos ();

    cmov (100.0, 400.0, 0.0);   /*  position to draw scoreboard */
    charstr ("New York 3");
    cmov (100.0, 385.0, 0.0);
    charstr ("Boston   2");
}

/*  draw_base() has not changed since the last example.              */

draw_base (x, y, z)
float   x,
        y,
        z;
{
    pmv (x, y, z);
    rpdr (5.0, 0.0, 0.0);
    rpdr (0.0, 5.0, 0.0);
    rpdr (-5.0, 0.0, 0.0);
    pclos ();
}
```

Workshop 13: view13.c

```
/*  view13.c -- This program introduces an alternative view to the        */
/*  overhead (blimp) orthographic view.  After displaying the movement   */
/*  of the ball from overhead, the ball rests using the waitandswap()    */
/*  routine.                                                              */
/*  Then an instant replay of the movement is shown.  The second time,   */
/*  there is a different point of view (using polarview) and             */
/*  perspective foreshortening.  Notice that the ball becomes larger     */
/*  as it approaches.                                                     */
/*  outfieldimage() and waitandswap() are new routines.                  */
/*  processinput() is changed.                                            */
/*                                                                        */
/*  graphics library calls introduced:  perspective(), polarview()       */

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

/*  global variables for menus          */

int     menu,
        killmenu;

/*  the main() routine has not changed since the last example.  */

main () {
    initialize ();
    drawimage ();
    while (TRUE) {      /*  loop forever.  swapbuffers and process      */
        while (!qtest())/*  input while looping                         */
            swapbuffers();
        processinput();
    }
}

/*  the initialize() routine has not changed since the last example.    */

initialize () {
    int i;

    keepaspect (1, 1);
```

```
    winopen ("diamond");

    doublebuffer();      /* display mode to become double buffer */
    gconfig();           /* display mode change takes effect     */

    qdevice(LEFTMOUSE); /* queue the LEFTMOUSE device            */
    qdevice(RIGHTMOUSE);/* queue the RIGHTMOUSE device           */
/*  queue the MOUSEX and MOUSEY devices, and tie them to         */
/*  the LEFT MOUSE                                               */
    tie (LEFTMOUSE, MOUSEX, MOUSEY);
    qdevice(REDRAW);     /* queue the REDRAW device              */

    mapcolor (8, 240, 240, 240);/*  make color of arcs, lines   */
    mapcolor (9, 0, 175, 0);    /*  make color of grass         */
    mapcolor (16, 240, 150, 0);  /*  make overlay (ball)         */
    mapcolor (24, 240, 150, 0);  /*  for colors 16, 24, and 25   */
    mapcolor (25, 240, 150, 0);

    killmenu = defpup ("Do you want to exit? %t|yes|no");
    menu = newpup ();
    addtopup (menu, "Baseball %t|exit program");
}

/*  If a point is chosen, the ball travels twice.  First, it     */
/*  is viewed with a familiar overhead view.  waitandswap()      */
/*  momentarily fixes the position of the ball.  Then the        */
/*  instant replay is viewed from a different location.          */

processinput() {
    short val;
    short mx, my;
    int dev;
    int domove;
    int menuval, kmenuval;

    domove = FALSE;
    while (qtest()) {
        dev = qread(&val);
        switch (dev) {
        case LEFTMOUSE:
            if (val == 1) /*  when mouse pressed, will do motion  */
                domove = TRUE;
            break;
        case MOUSEX:
            mx = val;
            break;
        case MOUSEY:
            my = val;
            break;
        case RIGHTMOUSE:
            if (val == 1) {
                menuval = dopup(menu);
                if (menuval == 1) {  /*  if exit chosen, ask user */
```

```
                    kmenuval = dopup (killmenu); /*  to reconfirm */
                    if (kmenuval == 1)
                        exit (0);
                }
            }
            break;
        case REDRAW:
            reshapeviewport();
            drawimage();
            break;
        default:
            break;
        }
    }
    if (domove) {
        drawimage();            /*  first view          */
        moveball(mx, my);
        waitandswap();          /*  freeze ball         */
        outfieldimage();        /*  instant replay      */
        moveball(mx, my);
        waitandswap();          /*  freeze ball         */
        drawimage();
    }
}

/*  the drawimage() routine has not changed since the last example.     */

drawimage () {
    ortho (0.0, 425.0, 0.0, 425.0, -10000.0, 10000.0);
    frontbuffer(TRUE);  /*  draw the field twice, once to each buffer   */
    color(BLACK);
    clear();
    diamond();
    frontbuffer(FALSE);
}

/*  outfieldimage() is introduced.  It specifies the alternate          */
/*  view for the instant replay.  The parameters for perspective()      */
/*  and polarview() were largely selected by good trial and error.      */

/*  The field of view was chosen to be 135 degrees, or   67 degrees to  */
/*  either side. The aspect ratio matches the aspect ratio of the       */
/*  window, namely 1. The near and far clipping planes clip only objects*/
/*  extremely close (within 0.1) or extremely far (beyond 10000.0).     */

/*  The polarview parameters set our view position 400 units from the   */
/*  origin, with a azimith angle of 135 degrees and an inclination of   */
/*  80 degrees producing the view seen from the outfield. The last      */
/*  parameter, twist, is set to zero so that the field remains upright. */

outfieldimage () {

    perspective (1350, 1.0, 0.1, 10000.0);
```

```
    polarview (400.0, 1350, 800, 0);
    frontbuffer(TRUE);  /*  draw the field twice, once to each buffer   */
    color(BLACK);
    clear();
    diamond();
    frontbuffer(FALSE);
}

/*  waitandswap() is introduced.  It causes the program to idle */
/*  between views of motion.  When idling, the program swaps    */
/*  buffers, which is vital when double buffer programs are     */
/*  running in the window manager.  The gsync() call delays     */
/*  execution until the next screen refresh cycle.              */

waitandswap () {
    int i;

    for (i = 0; i < 60; i++) {
        swapbuffers();
        gsync();
    }
}

/*  moveball() has not changed since the last example.          */

moveball(mx, my)
short mx, my;
{
    int i;
    int screenx, screeny;       /*  position of lower left corner        */
    int sizex, sizey;           /*  size of window                       */
    int distx, disty;           /*  distance ball will travel            */
    float rdistx, rdisty;       /*  relative distance ball will travel   */
    float incrx, incry;         /*  amount to move ball in 1 iteration   */
    float newx, newy;           /*  location for ball in flight          */

    getorigin (&screenx, &screeny); /* coordinates of lower left corner */
    getsize (&sizex, &sizey);       /* size of window                   */
    distx = mx - screenx;           /* distance to x mouse position     */
    disty = my - screeny;           /* distance to y mouse position     */

    /* This stuff calculates the relative position of the ball in the    */
    /* window as defined by the window size and projection.              */
    rdistx = (float) distx * 425.0 / (float) sizex;
    rdisty = (float) disty * 425.0 / (float) sizey;

    incrx = (float) rdistx / 50.0; /*  do 50 iterations         */
    incry = (float) rdisty / 50.0;
    newx = 0.0;
    newy = 0.0;
    writemask(16);       /*  protect the first four bitplanes               */
    for (i = 0; i < 50; i = i + 1) {
        newx = newx + incrx;
```

```
        newy = newy + incry;
        color (BLACK);          /*  clear away old ball                  */
        clear ();
        color (16);
        pushmatrix ();
        translate (newx, newy, 0.0);/*  translate local axes to newx, newy*/
        draw_ball ();
        popmatrix ();
        swapbuffers();
    }
        /*  Draw the ball into both front and back buffers.     */
    frontbuffer (TRUE); /*  draw final resting place of ball to         */
    color (BLACK);      /*  both buffers                                */
    clear ();
    color (16);
    pushmatrix ();
    translate (newx, newy, 0.0);/*  translate local axes to newx, newy  */
    draw_ball ();
    popmatrix ();
    frontbuffer(FALSE);
    writemask(0xfff);   /*  unprotect all bitplanes             */
}

/*  draw_ball() has not changed since the last example.         */

draw_ball ()
{
    circ (0.0, 0.0, 3.0);
    rotate (900, 'y');
    circ (0.0, 0.0, 3.0);
    rotate (600, 'x');
    circ (0.0, 0.0, 3.0);
    rotate (600, 'x');
    circ (0.0, 0.0, 3.0);
}

/*  diamond() has not changed since the last example.           */

diamond () {
    color (9);          /*  make color 9 current color  */
    arcf (0.0, 0.0, 375.0, 0, 900);     /*  grass       */
    color (8);          /*  make color 8 current color  */
    linewidth(2);       /*  change thickness of lines   */
    arc (0.0, 0.0, 375.0, 0, 900);      /*  fences      */
    arc (0.0, 0.0, 150.0, 0, 900);      /*  infield     */

    move (0.0, 0.0, 0.0);       /*  foul lines          */
    draw (0.0, 400.0, 0.0);
    move (0.0, 0.0, 0.0);
    draw (400.0, 0.0, 0.0);
    linewidth(1);       /*  restore thickness of lines  */
```

```
    circf (43.0, 43.0, 10.0);   /*  pitcher's mound      */

    draw_base (90.0, 0.0, 0.0); /*  first, second and third bases        */
    draw_base (90.0, 90.0, 0.0);
    draw_base (0.0, 90.0, 0.0);

    pmv (0.0, 0.0, 0.0);        /*  draw home plate      */
    pdr (0.0, 3.0, 0.0);
    pdr (3.0, 6.0, 0.0);
    pdr (6.0, 3.0, 0.0);
    pdr (3.0, 0.0, 0.0);
    pclos ();

    cmov (100.0, 400.0, 0.0);   /*  position to draw scoreboard */
    charstr ("New York 3");
    cmov (100.0, 385.0, 0.0);
    charstr ("Boston   2");
}

/*  draw_base() has not changed since the last example.        */

draw_base (x, y, z)
float   x,
        y,
        z;
{
    pmv (x, y, z);
    rpdr (5.0, 0.0, 0.0);
    rpdr (0.0, 5.0, 0.0);
    rpdr (-5.0, 0.0, 0.0);
    pclos ();
}
```

Workshop 14: parabola14.c

```
/*  parabola14.c -- parabola14 is similar to the previous workshop        */
/*  except that the ball leaves the surface of the playing field.         */
/*  The ball travels in a parabola from home plate to the selected        */
/*  point on the field.  This really gives the illusion of 3D.            */
/*  No new graphics commands introduced here.                             */
/*  parabola() is a completely new procedure.                             */

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

/*  global variables for menus          */

int     menu,
        killmenu;

/*  the main() routine has not changed since the last example.  */

main () {
    initialize ();
    drawimage ();
    while (TRUE) {      /*  loop forever.  swapbuffers and process      */
        while (!qtest())/*  input while looping                         */
            swapbuffers();
        processinput();
    }
}

/*  the initialize() routine has not changed since the last example.     */

initialize () {
    int i;

    keepaspect (1, 1);
    winopen ("diamond");

    doublebuffer();      /* display mode to become double buffer */
    gconfig();           /* display mode change takes effect     */

    qdevice(LEFTMOUSE); /* queue the LEFTMOUSE device           */
```

```
    qdevice(RIGHTMOUSE);/* queue the RIGHTMOUSE device          */
/*  queue the MOUSEX and MOUSEY devices, and tie them to        */
/*  the LEFT MOUSE                                              */
    tie (LEFTMOUSE, MOUSEX, MOUSEY);
    qdevice(REDRAW);    /* queue the REDRAW device              */

    mapcolor (8, 240, 240, 240);/*  make color of arcs, lines   */
    mapcolor (9, 0, 175, 0);    /*  make color of grass         */
    mapcolor (16, 240, 150, 0);  /*  make overlay (ball)        */
    mapcolor (24, 240, 150, 0);  /*  for colors 16, 24, and 25  */
    mapcolor (25, 240, 150, 0);

    killmenu = defpup ("Do you want to exit? %t|yes|no");
    menu = newpup ();
    addtopup (menu, "Baseball %t|exit program");
}

/*  processinput has not changed since the last example.         */

processinput() {
    short val;
    short mx, my;
    int dev;
    int domove;
    int menuval, kmenuval;

    domove = FALSE;
    while (qtest()) {
        dev = qread(&val);
        switch (dev) {
        case LEFTMOUSE:
            if (val == 1) /*  when mouse pressed, will do motion  */
                domove = TRUE;
            break;
        case MOUSEX:
            mx = val;
            break;
        case MOUSEY:
            my = val;
            break;
        case RIGHTMOUSE:
            if (val == 1) {
                menuval = dopup(menu);
                if (menuval == 1) {  /*  if exit chosen, ask user */
                    kmenuval = dopup (killmenu); /*  to reconfirm */
                    if (kmenuval == 1)
                        exit (0);
                }
            }
            break;
        case REDRAW:
            reshapeviewport();
            drawimage();
```

```
            break;
        default:
            break;
        }
    }
    if (domove) {
        drawimage();            /*  first view          */
        moveball(mx, my);
        waitandswap();          /*  freeze ball         */
        outfieldimage();        /*  instant replay      */
        moveball(mx, my);
        waitandswap();          /*  freeze ball         */
        drawimage();
    }
}

/*  the drawimage() routine has not changed since the last example.     */

drawimage () {
    ortho (0.0, 425.0, 0.0, 425.0, -10000.0, 10000.0);
    frontbuffer(TRUE);  /*  draw the field twice, once to each buffer   */
    color(BLACK);
    clear();
    diamond();
    frontbuffer(FALSE);
}

/*  outfieldimage() has not changed since the last example.             */

outfieldimage () {

    perspective (1350, 1.0, 0.1, 10000.0);
    polarview (400.0, 1350, 800, 0);
    frontbuffer(TRUE);  /*  draw the field twice, once to each buffer   */
    color(BLACK);
    clear();
    diamond();
    frontbuffer(FALSE);
}

/*  waitandswap() has not changed since the last example.               */

waitandswap () {
    int i;

    for (i = 0; i < 60; i++) {
        swapbuffers();
        gsync();
    }
}

/*  moveball() has been modified.  The initial position of the  */
/*  ball (0.0, 0.0), and the relative distance the ball will    */
```

```
/*  travel.  I say "relative" because the distance depends on    */
/*  how large a window is made.  The parabola() routine is       */
/*  called to draw the moving ball.                              */

moveball(mx, my)
short mx, my;
{
    int i;
    int screenx, screeny;       /*  position of lower left corner        */
    int sizex, sizey;           /*  size of window                       */
    int distx, disty;           /*  distance ball will travel            */
    float rdistx, rdisty;       /*  relative distance ball will travel   */

    getorigin (&screenx, &screeny); /* coordinates of lower left corner */
    getsize (&sizex, &sizey);       /* size of window                   */
    distx = mx - screenx;           /* distance to x mouse position     */
    disty = my - screeny;           /* distance to y mouse position     */

    /* This stuff calculates the relative position of the ball in the   */
    /* window as defined by the window size and projection.             */
    rdistx = (float) distx * 425.0 / (float) sizex;
    rdisty = (float) disty * 425.0 / (float) sizey;

    parabola(0.0, 0.0, rdistx, rdisty, 100.0, 100);
}

/*  Given initial conditions, the parabola() routine draws      */
/*  the moving ball.  (xstart, ystart) are the coordinates      */
/*  of the starting position of the parabola, assuming z = 0.   */
/*  (xdone, ydone) is where the ball will meet the ground       */
/*  again.  At the apex, the parabola reaches a height of       */
/*  zmax units.  The position of the ball will be calculated    */
/*  the number of times specified by the iterates variable.     */

parabola(xstart, ystart, xdone, ydone, zmax, iterates)
float xstart, ystart, xdone, ydone, zmax;
int iterates;
{
    float t, x, y, z;
    float tincr;

    writemask(16);      /*  protect the first four bitplanes            */
    tincr = 1.0 / (float) iterates;

    for (t = 0.0; t <= 1.0; t = t + tincr) {
        color(BLACK);
        clear();
        color (16);
        x = (xdone - xstart) * t;     /* velocity constant in x direction   */
        y = (ydone - ystart) * t;     /* velocity constant in y direction   */
        z = 4 * zmax * (t * (1 - t)); /* formula for z motion is a parabola */
        pushmatrix ();
        translate (x, y, z);
```

```
        draw_ball ();          /* draw ball at new position          */
        popmatrix ();
        swapbuffers();
    }
        /*  Draw the ball into both front and back buffers.    */
    frontbuffer (TRUE); /*  draw final resting place of ball to        */
    color (BLACK);      /*  both buffers                               */
    clear ();
    color (16);

    pushmatrix ();
    translate (x, y, z);
    draw_ball ();          /* draw ball at new position          */
    popmatrix ();
    frontbuffer(FALSE);
    writemask(0xfff);   /*  unprotect all bitplanes              */
}

/*  draw_ball() has not changed since the last example.         */

draw_ball ()
{
    circ (0.0, 0.0, 3.0);
    rotate (900, 'y');
    circ (0.0, 0.0, 3.0);
    rotate (600, 'x');
    circ (0.0, 0.0, 3.0);
    rotate (600, 'x');
    circ (0.0, 0.0, 3.0);
}

/*  diamond() has not changed since the last example.           */

diamond () {
    color (9);          /*  make color 9 current color  */
    arcf (0.0, 0.0, 375.0, 0, 900);     /*  grass        */
    color (8);          /*  make color 8 current color  */
    linewidth(2);       /*  change thickness of lines   */
    arc (0.0, 0.0, 375.0, 0, 900);      /*  fences       */
    arc (0.0, 0.0, 150.0, 0, 900);      /*  infield      */

    move (0.0, 0.0, 0.0);       /*  foul lines          */
    draw (0.0, 400.0, 0.0);
    move (0.0, 0.0, 0.0);
    draw (400.0, 0.0, 0.0);
    linewidth(1);       /*  restore thickness of lines  */
```

```
    circf (43.0, 43.0, 10.0);    /*  pitcher's mound     */

    draw_base (90.0, 0.0, 0.0); /*  first, second and third bases        */
    draw_base (90.0, 90.0, 0.0);
    draw_base (0.0, 90.0, 0.0);

    pmv (0.0, 0.0, 0.0);         /*  draw home plate     */
    pdr (0.0, 3.0, 0.0);
    pdr (3.0, 6.0, 0.0);
    pdr (6.0, 3.0, 0.0);
    pdr (3.0, 0.0, 0.0);
    pclos ();

    cmov (100.0, 400.0, 0.0);    /*  position to draw scoreboard */
    charstr ("New York 3");
    cmov (100.0, 385.0, 0.0);
    charstr ("Boston   2");
}

/*  draw_base() has not changed since the last example.         */

draw_base (x, y, z)
float   x,
        y,
        z;
{
    pmv (x, y, z);
    rpdr (5.0, 0.0, 0.0);
    rpdr (0.0, 5.0, 0.0);
    rpdr (-5.0, 0.0, 0.0);
    pclos ();
}
```

Glossary

We used two sources for definitions of standard graphics terminology: William M. Newman and Robert F. Sproull, *Principles of Interactive Graphics*, (1979) and James D. Foley and Andries Van Dam, *Fundamentals of Interactive Graphics*, (1982).

aspect ratio: The ratio of the height of an object to its width. A rectangle of width ten inches and height five inches has an aspect ratio of 10/5 or 2.

asynchronous: Not synchronized in time. For example, input events occur at the whim of the user—the program may read them later.

attribute: A model feature in the graphics. If the color is set to "RED", it will remain red until changed, and everything that is drawn will be drawn in red. Color is an attribute. Other attributes include linestyle, linewidth, texture, pattern, and font.

azimuthal angle: If an object is sitting on the ground, with its z coordinate straight up, the azimuthal viewing angle is the angle the observer makes with the y axis in the x-y plane. If the observer walks in a circle with the object at the center, the azimuthal angle is the only thing that varies.

B-spline: A cubic spline approximation to a set of four control points having the property that slope and curvature are continuous across sets of control points.

basis: In the Graphics Library, a curve or patch basis is a 4x4 matrix that controls the relationship between control points and the approximating spline. B-splines, Bezier curves, and Cardinal splines all differ in that they have different bases. If B is a basis, and P_1, P_2, P_3, and P_4 are 4D control points, then the spline determined by them is given by:

Bezier: A cubic spline approximation to a set of four control points that passes through the first and fourth control points, and has a continuous slope where two spline segments meet.

bitplanes: A bitplane supplies one bit of color information per pixel on the display. Thus, an eight bitplane system allows 2^8 different colors to be displayed at each pixel.

Boolean: A value of TRUE or FALSE. TRUE=1 and FALSE=0

bounding box: A rectangle (2D) that bounds an object. A bounding box is used to determine whether the object lies inside a clipping region. See **clipping**.

button: Buttons on the IRIS include those on the keyboard, mouse, lightpen, or buttons on the dial and button box.

Cardinal spline: A cubic spline whose endpoints are the second and third of four control points. A series of cardinal splines will have a continuous slope, and will pass through all but the first and last control points.

clipping: If an object overlaps the boundaries of a window, it is *clipped*. The part of an object that appears in the window is displayed and the rest is ignored. See **window**.

clipping planes: Before clipping occurs, object space is mapped to normalized viewing coordinates in homogeneous coordinates. Each of the clipping Geometry Engines clips to one of the clipping planes $x=\pm w$; $y=\pm w$; or $z=\pm w$. These clipping planes correspond to the left, right, top, bottom, near, and far planes bounding the viewing frustum.

clipping subsystem: Four or six Geometry Engines that clip to four or six of the clipping planes.

color map: A hardware map that associates eight bits each of red, green, and blue with bit combinations from the bitplanes. There are, therefore, $2^{24} \approx 16{,}700{,}000$ colors, but on an eight bitplane system, for example, only $2^{8} \approx 256$ of these colors would be visible at once. The color map determines which ones these will be.

concatenation: In the Graphics Library, concatenation refers to combining a series of geometric transformations—rotations, translations, scaling, etc. Concatenation of transformations corresponds to matrix multiplication.

control points: Three-dimensional points that control the shape of an approximating spline curve. The IRIS provides hardware support for rational cubic splines, the specifications of which requires four control points.

culling: If an object is smaller than the minimum size specified in the command, it is *culled*: no further commands in the object are interpreted. See **clipping, pruning.**

current character position: the two-dimensional screen coordinates where the next character string or pixel read/write will occur.

current color: The color in which geometry will be drawn.

current graphics position: The homogeneous three-dimensional point from which geometric drawing commands will draw. The current graphics position is not necessarily visible.

current transformation matrix: The transformation matrix on top of the matrix stack. All points passed through the Geometry Pipeline are multiplied by the current transformation matrix before being passed on to the clipping and scaling subsystem.

cursor: A graphical object such as an arrowhead which can be moved about the screen by means of an input device (typically a mouse).

cursor glyph: a 16x16 raster pattern that determines the shape of the cursor.

depth-cueing: Varying the intensity of a line with z-depth. Typically, the points on the line further from the eye are darker, so the line seems to fade into the distance.

device: A valuator, button, or the keyboard. Buttons have values of 0 or 1 (up or down); valuators return values in a range, and the keyboard returns ASCII values.

dial and switch box: An I/O device with eight dials (valuators) and thirty-two switches.

display list: A data structure that contains a sequence of compiled commands. This intermediate form can be quickly and efficiently converted into hardware commands.

double buffer mode: A mode in which two buffers are alternately displayed and updated. A new image can be created in the buffer while the previous image is being displayed. In double buffer mode, colors are taken from the **color map**. See **color map, RGB mode, single buffer mode.**

event queue: A queue that records changes in input devices—buttons, valuators, and the keyboard. The event queue provides a time-ordered list of input events.

eye coordinates, eye space: The coordinate system whose origin lies at the viewpoint. See **object space, screen space, world space**.

feedback: Geometrical data that is passed through some or all of the clipping, scaling, and matrix subsystems and is returned to the applications processor.

field of view: The extent of the area which is under view. The field of view is defined by the **viewing object** in use.

fine clipping: Fine clipping masks all drawing commands to a rectangular region on the screen. It would be unnecessary except for the case of character strings. The origin of a character string after transformation may be clipped out, and the string would not be drawn. By doing gross clipping with the viewport and fine clipping with the screen masks, strings can be moved smoothly off the screen to the left or bottom. See **gross clipping.**

font: A set of characters in a particular style. See **raster font, object font.**

forward difference matrix: A 4x4 matrix that is interacted by adding each row to the next and the bottom row is output as the next point. Points so generated generally fall on a rational cubic curve.

front and back buffers: In double buffer mode, the bitplanes are separated into two sets—the front and the back buffers. Bits in front buffer planes are visible and those in the back are not. Typically, an application draws into the back buffer and views the front buffer for dynamic graphics.

gamma correction: A logarithmic assignment of intensities to color map entries for shading applications. This is required since the human eye perceives intensities logarithmically rather than linearly.

Geometry Accelerator: A custom VLSI chip that provides floating-point conversion and buffering for the Geometry Pipeline.

Geometry Engine: A custom VLSI chip that performs matrix multiplication, clipping, or scaling in 3D homogeneous coordinates. There are 10 or 12 Geometry Engines in the Geometry Pipeline.

Geometry Pipeline: A combination of Geometry Accelerators and Geometry Engines that does geometric transformations, clipping, and scaling in the IRIS.

Gouraud shading: A method to shade polygons smoothly based on the intensities at their vertices. The color index is uniformly interpolated along each edge, and then the edge values are uniformly interpolated along each scan line. For realistic shading, the colors associated with the color indices should be gamma-corrected.

graphical object: A set of drawing commands and transformation commands compiled into a user-defined object. This object can then be drawn with a single command.

gross clipping: Clipping done by the viewport. It is the same (typically) as fine clipping except in the case of character strings. See **fine clipping**.

hidden surface: A surface of a geometric object that is not visible because it is obscured by other surfaces. See **z-buffering**.

hitcode: In picking or selecting mode, an object can be clipped against any combination of the 6 (or 4) clipping planes. The hitcode is a 6-bit number that indicates the planes against which clipping occurred.

immediate mode: The IRIS is in immediate mode except while an object is being edited. In this mode, graphics commands are executed immediately rather than being compiled into a graphical object.

incidence angle: In the command the incidence angle is the angle the viewport makes with the z axis in the z-y plane.

instantiate: To make an instance of. To replicate.

line of sight: A line which extends from the viewpoint through a point specified by the user. The line of sight defines the direction of view.

linear interpolation: Linear interpolation approximates an output value for an input between two other inputs whose exact output values are known. If a and b are input values whose outputs are A and B, respectively, and a<c<b, then the approximate output value of c given by linear interpolation is:

linestyle: he pattern used to draw a line. A linestyle might be solid or broken into a pattern of dashes.

linewidth: The width of a line in pixels.

matrix stack: A stack of matrices on the IRIS with hardware and software support. The top matrix on the stack is the active matrix, and all points passed through the Geometry Pipeline are multiplied by that matrix.

matrix multiplier: The first four Geometry Engines in the Pipeline form a hardware matrix multiplier unit. An input matrix is multiplied by the matrix on top of the matrix stack.

mirroring: The creation of a mirror image of an object.

modeling: The representation of a real object as a graphical object in world space.

name stack: A stack of 16-bit names kept by the raster subsystem and used to locate hits in display lists during picking and selecting operations.

NTSC: A video display format supported by the IRIS that is used for broadcast style pictures.

null-terminated: Having a zero byte at the end. In the C language, character strings are stored this way internally.

object: A graphical representation in world space which is composed of a series of primitive graphical units such as line segments, points, characters, or text strings. The object is compiled into a display list and can be instantiated in different orientations and sizes through appropriate use of modeling.

object font: A font in which characters are defined as graphical objects. Like all other graphical objects, object font characters can be scaled and rotated. See **raster font.**

object coordinates, object space: The space in which a graphical object is defined. A convenient point is chosen as the origin and the object is defined relative to this point. See **eye space, screen space, world space.**

orthographic projection: A representation in which the lines of projection are parallel. Orthographic projections lack *perspective foreshortening* and its accompanying sense of depth and realism. Because they are simple to draw, orthographic projections are often used by draftsmen. See **perspective projection.**

parametric cubic curve: A curve defined by the equation:

where x, y, z, and w are cubic polynomials. t is the parameter and typically varies between 0 and 1.

patch: A local homeomorphism of the Euclidean plane.

pattern: A 16x16, 32x32, or 64x64 array of bits defining the texturing of polygons on the IRIS display.

perspective projection: Perspective projection is a technique used to achieve realism when drawing objects. In a perspective projection, the lines of projection meet at the viewpoint; thus the size of an object varies inversely with its distance from the source of projection. The farther an object or part of an object is from the viewer, the smaller it will be drawn. This effect, known as *perspective foreshortening*, is similar to the effect achieved by photography and by the human visual system. See **orthographic projection.**

picking: Determining the geometric objects that lie at or near the cursor on the display screen.

picking region: A region around the cursor origin in which objects will be picked if they appear there.

pixel: A rectangular picture element. A display screen is composed of an array of pixels. In a black-and-white system, pixels are turned on and off to form images. In a color system, each pixel has three components: red, green, and blue. The intensity of each component can be controlled.

polar coordinates: A coordinate system in the plane related to the standard Cartesian (x,y) coordinates by the following equations:

r and Θ are the polar coordinates of the point.

polled i/o devices: Devices whose current values are read by the user process.

pre-multiplication: Matrix multiplication on the left. If a matrix M is pre-multiplied by a matrix T, the result is TM.

precision: The number of straight line segments used to approximate one segment of a spline.

pruning: Eliminating the drawing of parts of the display list because a bounding box test shows that they are not visible.

queued i/o devices: Devices whose changes are recorded in the event queue.

raster font: A font in which the characters are defined directly by a raster bit map. See **font, object font.**

raster subsystem: That part of the IRIS concerned with geometry after it has been transformed and scaled to screen coordinates. It includes scan conversion and recording.

refresh rate: The rate at which a monitor is refreshed. A 60 Hz monitor is redrawn 60 times per second.

relative drawing commands: Commands that draw relative to the current graphics position as opposed to being drawn at absolute locations.

RGB mode: A mode in which colors are directly controlled by the user without using a **color map**. Images are simultaneously updated and displayed. See **color map, double buffer mode, single buffer mode.**

RGB value: The set of red, green, and blue intensities that compose a color is that color's **RGB value.**

right-hand rule: If the right hand is wrapped around the axis of rotation, the fingers curl in the same direction as positive rotation.

rotation: The transformation of an object by rotating it about an axis. See **transformation.**

scaling: Uniform stretching of an object along an axis.

scaling subsystem: The final two Geometry Engines in the Geometry Pipeline scale normalized world coordinates in the range -1≤ x,y,z ≤) to physical screen coordinates.

screen coordinates, screen space: The coordinate system that defines the display screen. The screen on the IRIS measures 1024 pixels wide by 768 pixels high, with the origin in the lower left corner. See **eye space, object space, world space.**

screenmask: A rectangular area of the screen that all drawing operations are clipped to. It is normally set equal to the viewport.

selecting: A mechanism for finding the objects lying in an arbitrary rectangle on the screen. See **picking.**

selecting region: A rectangle on the display screen. Objects drawn within this region will be reported as hits when the command is used. See **picking.**

short: A 16-bit integer.

single buffer mode: A mode in which images are simultaneously displayed and updated in a single buffer. Colors are taken from a color map. See **double buffer mode, RGB mode.**

swap interval: The number of retrace intervals that go by between swapping the front and back buffers.

tags: Markers in the display list used as locations for object editing.

textport: A region on the display screen used to present textual output from graphical or non-graphical programs.

texture: A pattern used to fill rectangles, convex polygons, arcs, and circles.

transformations: Transformations change the size and orientation of an object under view by changing the object itself or the position of the viewpoint. Standard mathematical techniques like coordinate geometry, trigonometry, and matrix methods are used to compute the new position of the object after the transformation. Modeling transformations change the position or orientation of the *object*. Viewing transformations change the position, size, or orientation of the *viewpoint*. Projection transformations redefine the boundaries of the *clipping region*.

twist: A rotation around the line of sight.

valuator: An input/output device that returns a value in a range. For example, a mouse is logically two valuators—the x position and the y position.

vector product: Another term for the vector cross product. If $a = (a_1, a_2, a_3)$ and $b = (b_1, b_2, b_3)$ are two 3D vectors, the vector product $a \times b = (a_2 b_3 - b_2 a_2, a_3 b_1 - b_3 a_1, a_1 b_2 - b_1 a_2)$.

vertical retrace period: On a 30Hz interlaced display, it is 1/30 second.

viewing object: A graphical object which defines the region of **world space** which can be seen on the display screen. See **world space, field of view.**

viewpoint: The center of the **field of view**. The **viewpoint** is the origin of both the **line of sight** and the **eye coordinate system**. See **eye space, field of view, line of sight, viewing object**.

viewport: A rectangle on the screen in which the contents of the window are displayed. See **window**.

window: A truncated pyramid in **world space** which the user defines and which is mapped to the display screen. See **viewport**.

world coordinates, world space: The user-defined coordinate system in which an image is described. Modeling commands are used to position graphical objects in world space. Viewing and projection transformations define the mapping of world space to screen space. See **modeling, object space, screen space, transformations**.

writemask: A bit mask that controls write access to the bitplanes. During any drawing operation, only those planes enabled by a "1" in the bit mask can be altered.

z-buffering: A method for hidden surface removal that keeps track (for each pixel) of the distance from that point to the eye. Additional geometry is drawn over that pixel only if it is closer to the eye that what was previously there. The IRIS supports such a z-buffer in the raster subsystem.

Index

A
addtopup, W-31; W-33
applications/graphics processor, 10-3
arc, 3-16
arcf, 3-16
arcs, 3-14–3-16
  arc, 3-16
  arcf, 3-16
  arcs (fig), 3-15
attach, W-49
attachcursor, 7-5
attached window, W-2
available bitplanes, initial value, 2-2

B
backbuffer, 6-5
backface, 12-7
backface removal, example program
  in C, 12-8–12-10; example
  program in FORTRAN, 12-10–12-13
backfacing polygon removal, 12-6–12-13
  backface, 12-7
  example, 12-8–12-13
  getbackface, 12-7
  overview, 12-6
background menu (fig), W-3; W-5
bbox2, 8-7–8-8
Bezier cubic curve, 11-3–11-4; (fig) 11-5;
  example program in C, 11-20–11-21
bindable functions, W-49
bindcolor, W-48
bindfunc, W-48
bindindex, W-48
binding
  colors to pop-up menus and cursor, W-53
  colors to title bar and
    borders, W-51–W-52
  function to a menu entry, W-40
  function to a whole menu, W-39–W-40
bitplanes and color maps (fig) 6-2
blankscreen, 6-25
blink, 6-11–6-12
blinking, initial value, 2-2
blkqread, 7-10–7-11
blop, example program in C, W-43–W-46
blue box, example program in C, 3-6;
  example program in FORTRAN, 3-6–3-7
blue rectangle, example program in C, 6-13;
  example program in FORTRAN, 6-13;
  in RGB mode, 6-14–6-15
bounding boxes (fig), 8-9
B-spline cubic curve (fig) 11-5; 11-6–11-7
button events, interpreting, W-48–W-51

C
C, documentation conventions, 1-5
C, example programs
  backface removal, 12-8–12-10
  Bezier curve, 11-20–11-21
  blop, W-43–W-46
  blue box, 3-6
  blue rectangle, 6-13;
    in RGB mode, 6-14–6-15
  crash, 9-17–9-19
  curve segments, 11-12–11-13
  depth-cued cube, 13-8–13-10
  double buffer mode, W-27–W-29
  feedback, 10-8–10-10
  feedback and depthcue, 10-10–10-11
  joined curve segments, 11-16–11-17
  picking, 9-10–9-12
  redraw, W-59–W-64

shaded polygon, 13-4–13-5
single buffer mode, W-25–W-26
square drawn with points, 3-4
surface patches, 11-28–11-30
text, 3-18
xfpt, 10-13–10-14
z-buffer, 12-4–12-5
callfunc, 8-18
callobj, 8-5–8-7
Cardinal spline cubic curve, 11-4–11-6; (fig), 11-5
cedit, example from, W-41–W-43
character definition, sample (fig), 5-10
charstr, 3-17
chunksize, 8-17–8-18
circ, 3-13–3-14
circf, 3-14
circles, 3-13–3-14
circ, 3-13–3-14
circf, 3-14
clear, 3-3
clearhitcode, 9-9
clipping, fine and gross, 3-17; 4-18; (fig), 4-19
clkoff, 7-12
clkon, 7-11
closeobj, 8-2
cmov, 3-16–3-17
color, 6-12; initial value, 2-2
color editing program, example from *cedit*, W-41–W-43
color map mode, initial value, 2-2
color maps, 6-8–6-12
blink, 6-11–6-12
cyclemap, 6-11
entries, setting, W-53
getmap, 6-10
getmcolor, 6-9
getmmode, 6-10
mapcolor, 6-8
multimap, 6-9–6-10
onemap, 6-9
overview, 6-8
setmap, 6-10
color modes, see display and color modes
colors and writemasks, 6-12–6-19
color, 6-12
colors, 6-12–6-15
example program, 6-13; 6-14–6-15
getcolor, 6-14
RGBcolor, 6-14
colors for drawing pop-up menus, W-37
command tokens and associated data, 10-6
compactify, 8-17
confirmation menu (fig), W-6
coordinate transformations, see transformations, coordinate
crash, example program in C, 9-17–9-19; example program in FORTRAN, 9-19–9-21
crv, 11-9–11-10
crvn, 11-15
culling, 8-7
curorigin, 6-22
current character position, 3-2
current graphics position, 3-2
cursoff, 6-24
curson, 6-24–6-25
cursor, W-2
binding colors to, W-53
devices, 7-15
initial value, 2-2
cursors, 6-19–6-25
blankscreen, 6-25
curorigin, 6-22
cursoff, 6-24
curson, 6-24–6-25
defcursor, 6-20
getcursor, 6-22
gRGBcursor, 6-23
overview, 6-19
RGBcursor, 6-23
sample cursors (fig), 6-21
setcursor, 6-20
curvebasis, 11-8
curveit, 11-19
curve mathematics, 11-2–11-7
Bezier cubic curve, 11-3–11-4
Bezier, Cardinal, and B-spline curves (fig), 11-5
B-spline cubic curve, 11-6–11-7
Cardinal spline cubic curve, 11-4–11-6
overview, 11-2–11-3
parametric cubic curve, 11-2
curveprecision, 11-8
curves and surfaces, 11-1–11-33
curve mathematics, 11-2–11-7

drawing curves, 11-7–11-22
drawing surfaces, 11-23–11-33
overview, 11-1
curves, drawing, 11-7–11-22
crv, 11-9–11-10
crvn, 11-15
curvebasis, 11-8
curveit, 11-19
curveprecision, 11-8
curve segments (fig), 11-11
defbasis, 11-8
example, 11-12–11-15; 11-16–11-19; 11-20–11-21
overview, 11-7
rational curves, 11-21–11-22
rcrv, 11-22
rcrvn, 11-22
curve segments
curve segments (fig), 11-11
example program in C, 11-12–11-13
example program in FORTRAN, 11-13–11-15
curves, rational, 11-21–11-22
customizing *mex*, W-47–W-55
bindable functions, W-49
binding colors to pop-up menus and the cursor, W-53
binding colors to the title bar and borders, W-51–W-52
button events, interpreting, W-48–W-51
color map entries, setting, W-53
default *.mexrc* configuration file, W-48
desktop, arranging, W-54–W-55
mexrc configuration file entries, W-48
.mexrc, sample, W-53–W-54
overview, W-47–W-48
title bar and border regions of a window, W-52
cyclemap, 6-11

D

dbtext, 7-13
default return values, W-36
default values, getting back for menu selections, W-38–W-39
defbasis, 11-8
defcursor, 6-20
defining a graphical object, 8-1–8-4
closeobj, 8-2
delobj, 8-4
genobj, 8-4
isobj, 8-4
makeobj, 8-2
object definition (fig), 8-3
deflinestyle, 5-2
defpattern, 5-7
defpup, W-31; W-32
defrasterfont, 5-9
delobj, 8-4
deltag, 8-12
depthcue, 13-6; mode, initial value, 2-2
depth-cued cube, example program in C, 13-8–13-10; example program in FORTRAN, 13-10–13-12
depth-cueing, 13-6–13-12
depthcue, 13-6
example, 13-8–13-12
getdcm, 13-7
shaderange, 13-7–13-8
.deskconfig, W-55
desktop, arranging, W-54–W-55
device.h, 7-1
device, initializing, see initializing a device
devices, other, 7-4
device, polling, see polling a device
devices, controlling peripheral input/output, see peripheral input/output devices, controlling
devices, special, see special devices
devport, 7-4
display and color modes, 6-1–6-25
color maps, 6-8–6-12
colors and writemasks, 6-12–6-19
cursors, 6-19–6-25
display modes, 6-1–6-7
overview, 6-1
display mode, 6-1–6-7
backbuffer, 6-5
bitplanes and color maps (fig), 6-2
doublebuffer, 6-4
finish, 6-7
frontbuffer, 6-6
gconfig, 6-3
getbuffer, 6-6
getdisplaymode, 6-6
getplanes, 6-7

gsync, 6-7
initial value, 2-2
overview, 6-1
RGBmode, 6-3
single buffer, 6-4
single buffer and double buffer modes, 6-3–6-7
swapbuffers, 6-5
swapinterval, 6-5
dopup, W-31; W-35
double buffer modes, see single buffer and double buffer modes;
double buffer mode, example program in C, W27–W-29
draw, 3-5; 10-6
drawing curves, see curves, drawing
drawing positions, current, 3-2
getcpos, 3-2
getgpos, 3-2
drawing, relative, 3-7
rdr, 3-7
rmv, 3-7
drawing routines, 3-1–3-21
arcs, 3-14–3-16
circles, 3-13–3-14
clearing viewport, 3-3
current drawing positions, 3-2
lines, 3-5–3-7
overview, 3-1
pixels, writing and reading, 3-19–3-21
points, 3-3–3-5
polygons, 3-10–3-13
rectangles, 3-8–3-10
text, 3-16–3-19
drawing surfaces, see surfaces, drawing

E

editing
color program, example from *cedit*, W-41–W-43
graphical objects, 8-1–8-18
object definition, 8-11
editobj, 8-8
endfeedback, 10-7
endfullscrn, W-20; W-22
endpick, 9-8
endpupmode, W-31; W-34
endselect, 9-15
event queue, 7-7–7-11; W-25
blkqread, 7-10–7-11
isqueued, 7-8
noise, 7-8–7-9
qdevice, 7-7
qenter, 7-9
qread, 7-10
qreset, 7-11
qtest, 7-10
tie, 7-9
unqdevice, 7-8
eye coordinate system, 4-1

F

feedback, 10-7
effect of clipping on (fig), 10-5
example program in C, 10-8–10-10; example program in FORTRAN, 10-9
feedback (fig), 10-2
feedback and depthcue, example program in C, 10-10–10-11; example program in FORTRAN, 10-11–10-12
feedback mode, 10-6–10-16
command tokens and associated data, 10-6
endfeedback, 10-7
examples, 10-8–10-12; 10-13–10-16
feedback, 10-7
passthrough, 10-7
xfpt, 10-12–10-13
finish, 6-7
font, 5-11; initial value, 2-2
fonts, 5-9–5-12; raster, W-24
defrasterfont, 5-9
font, 5-11
getdescender, 5-12
getfont, 5-11
getheight, 5-11
sample character definition (fig), 5-10
strwidth, 5-12
foreground, W-12; W-15–W-16
FORTRAN, documentation conventions, 1-5
FORTRAN, example programs
backface removal, 12-10–12-13
blue box, 3-6–3-7
blue rectangle, 6-13; in RGB mode, 6-15
crash, 9-19–9-21

curve segments, 11-13–11-15
depth-cued cube, 13-10–13-12
feedback, 10-9
feedback and depthcue, 10-11–10-12
joined curve segments, 11-17–11-19
picking, 9-12–9-14
shaded polygon, 13-5–13-6
square drawn with points, 3-4–3-5
surface patches, 11-30–11-32
text, 3-18–3-19
xfpt, 10-14–10-16
z-buffer, 12-5–12-6
freepup, W-31; W-36
frontbuffer, 6-6
fudge, W-12; W-14
fullscrn, W-20; W-22

G

gbegin, 2-3
gconfig, 6-3
genobj, 8-4
gentag, 8-12
geometry accelerators, 10-3
Geometry Engine, 1-1, 10-3, 10-6
Geometry Pipeline
about, 1-1; 10-1–10-6
effect of clipping on feedback (fig), 10-5
Geometry Pipeline (fig), 10-4
subsystems, 10-3
Geometry Pipeline feedback, 10-1–10-16
feedback mode, 10-6–
Geometry Pipeline, 10-1–10-6
getbackface, 12-7
getbuffer, 6-6
getbutton, 7-6
getcmmode, 6-10
getcolor, 6-14
getcpos, 3-2
getcursor, 6-22
getdcm, 13-7
getdepth, 12-3
getdescender, 5-12
getdev, 7-7
getdisplaymode, 6-6
getfont, 5-11
getgpos, 3-2
getheight, 5-11
gethitcode, 9-8
getlsbackup, 5-5
getlsrepeat, 5-6
getlstyle, 5-5
getlwidth, 5-6
getmap, 6-10
getmatrix, 4-23
getmcolor, 6-9
getmem, 8-18
getmonitor, 2-6
getopenobj, 8-10
getorigin, W-20; W-21
getothermonitor, 2-7
getpattern, 5-7
getplanes, 6-7
getport, W-11
getresetls, 5-6
getscrmask, 4-18
getshade, 13-4
getsize, W-20; W-21
gettp, 14-2
getvaluator, 7-6
getviewport, 4-17
getwritemask, 6-16
getzbuffer, 12-4
gexit, 2-6
gflush, 2-5
ghost devices, 7-15
gid, W-57
ginit, 2-3
global state attributes, 2-1–2-7
initialization, 2-1–2-7
initial values, 2-2
overview, 2-1
saving, 2-7
Gouraud shading, 13-1; (fig), 13-3
graphical objects, 8-1–8-18
defining, 8-1–8-4
editing, 8-8–8-18
overview, 8-1
using, 8-5–8-8
graphics hardware, IRIS (fig), 1-2
graphics initialization, W-24
Graphics Library, 1-3–1-4
graphics program, W-2; executing from *mex*, W-8
greset, 2-4
gRGBcursor, 6-23
gRGBmask, 6-19

gselect, 9-15
gsync, 6-7

H
hidden surfaces, see surfaces, hidden
hierarchical objects (fig), 8-6
hogmode, W-49
hogwhiledown, W-49

I
imakebackground, W-12; W-15
initialization of global state
attributes, 2-1–2-7
gbegin, 2-3
getmonitor, 2-6
getothermonitor, 2-7
gexit, 2-6
gflush, 2-5
ginit, 2-3
greset, 2-4
setfastcom, 2-5
setmonitor, 2-6
setslowcom, 2-5
initializing a device, 7-4–7-5
attachcursor, 7-5
devport, 7-4
setvaluator, 7-5
initnames, 9-7
input buttons, 7-3
INPUTCHANGE, 7-15
input devices, about, 7-1
input/output routines, 7-1–7-15
event queue, 7-7–7-11
initializing a device, 7-4–7-5
peripheral input/output devices, controlling, 7-11–7-14
polling a device, 7-6–7-7
polling and queuing, 7-1–7-4
special devices, 7-14–7-15
input valuators, 7-3
IRIS
documentation conventions, 1-5
Graphics Library, 1-3–1-4
IRIS graphics hardware (fig), 1-2
overview, 1-1–1-3
related publications, 1-6
ismex, W-20; W-23
isobj, 8-4
isqueued, 7-8
istag, 8-12

J
joined curve segments, example program in C, 11-16–11-17; example program in FORTRAN, 11-17–11-19
joined patches (fig), 11-33

K
keepaspect, W-12; W-13
keyboard devices, 7-14
kill, W-49

L
lampoff, 7-12
lampon, 7-12
lines, 3-5–3-7
draw, 3-5
example programs, 3-6–3-7
move, 3-5
relative drawing, 3-7
linestyle, initial value, 2-2
linestyle backup, initial value, 2-2
linestyles, 5-1–5-6
deflinestyle, 5-2
getlsbackup, 5-5
getlsrepeat, 5-6
getlstyle, 5-5
getlwidth, 5-6
getresetls, 5-6
linewidth, 5-5
lsbackup, 5-3
lsrepeat, 5-4
modifying linestyle pattern, 5-3–5-6
overview, 5-1
resetls, 5-4
setlinestyle, 5-2
linewidth, 5-5; initial value, 2-2
loadmatrix, 4-21
loadname, 9-7
.login, W-54–W-55
lookat, 4-10; (fig), 4-11
lsbackup, 5-3
lsrepeat, 5-4; initial value, 2-2

M
makeobj, 8-2
maketag, 8-10
mapcolor, 6-8; W-48
mapping screen coordinates to
  world coordinates, 9-1–9-2
  mapw, 9-1–9-2
  mapw2, 9-2
mapw, 9-1–9-2
mapw2, 9-2
maxsize, W-12; W-13
menu, W-49
menu button, W-50
menu formats, advanced, W-38–W-41
  binding a function to a menu entry, W-40
  binding a function to a whole
    menu, W-39–W-40
  default values, getting back for
    menu selections, W-38–W-39
  nested (rollover) menu,
    making, W-40–W-41
  return values, changing for menu
    selections, W-39
  summary, W-41
  title bar, making, W-39
menus, see individual listings
mex
  about, W-1
  attaching to, W-7
  background menu (fig), W-3
  creating windows, W-3–W-4
  customizing, W-47–W-55
  default configuration, W-1–W-8
  executing graphics programs from, W-8
  functions, W-3
  interacting with, W-4–W-6
  programming with, W-9–W-29
  terminology, W-2
  title font, selecting, W-7
.mexrc, W-34; W-36; W-37; W-47
  configuration file entries, W-48
  default configuration file, W-48
  sample, W-53–W-54
minsize, W-12
MODECHANGE, 7-15
modeling routine order (fig), 4-6
modeling routines (fig), 4-4
modeling transformations, see
  transformations, modeling
move, 3-5; 10-6; W-49
movegrow, W-49; W-50
multimap, 6-9–6-10
multmatrix, 4-22

N
nested (rollover) menu, making, W-40–W41
newpup, W-31; W-32
newtag, 8-13
noise, 7-8
noport, W-12; W-15

O
objdelete, 8-14
object coordinate system, 4-1
object editing, 8-8–8-18
  bounding boxes (fig), 8-9
  callfunc, 8-18
  chunksize, 8-17
  compactify, 8-17
  deltag, 8-12
  editing an object definition (fig), 8-11
  editobj, 8-8
  example, 8-15–8-16
  gentag, 8-12
  getmem, 8-18
  getopenobj, 8-10
  identifying display list items
    with tags, 8-10–8-13
  inserting, deleting, and replacing
    within objects, 8-13–8-15
  istag, 8-12
  maketag, 8-10
  newtag, 8-13
  objdelete, 8-14
  object memory management, 8-16–8-18
  objinsert, 8-13–8-14
  objreplace, 8-14–8-15
  overview, 8-8
object memory management, 8-16–8-18
objinsert, 8-13
objreplace, 8-14–8-15
onemap, 6-9
opening and closing windows, W-9–W-11
  getport, W-11
  winclose, W-11
  winopen, W-9–W-10

ortho, 4-15; (fig), 4-16
ortho2, 4-15

P

pagecolor, 14-3
pagewritemask, 14-3
parametric bicubic surface, 11-23
parametric cubic curve, 11-2
Pascal, documentation conventions, 1-5
passthrough, 10-6, 10-7
patch, 11-25
patchbasis, 11-24
patchcurves, 11-24
patches, see curves and surfaces
patches, joined (fig), 11-33
patchprecision, 11-25
pattern, initial value, 2-2
patterns, 5-6–5-8
  defpattern, 5-7
  getpattern, 5-7
  overview, 5-6
  sample patterns (fig), 5-8
  setpattern, 5-7
pclos, 3-13; 10-6
pdr, 3-12; 10-6
peripheral input/output devices,
  controlling, 7-11–7-14
  clkoff, 7-12
  clkon, 7-11
  dbtext, 7-13
  lampoff, 7-12
  lampon, 7-12
  ringbell, 7-13
  setbell, 7-13
  setdblight, 7-14
perspective, 4-12; (fig), 4-13
pick, 9-6
picking, 9-2–9-14
  clearhitcode, 9-9
  endpick, 9-8
  example program in C, 9-10–9-12;
    example program
    in FORTRAN, 9-12–9-14
  gethitcode, 9-8
  initnames, 9-7
  loadname, 9-7
  name stack, using, 9-6–9-9
  overview, 9-2–9-6
  pick, 9-6
  picking (fig), 9-4
  picking region, defining, 9-9
  picksize, 9-9
  popname, 9-7
  pushname, 9-7
  size, initial value, 2-2
picking and selecting, 9-1–9-21
  mapping screen coordinates to
    world coordinates, 9-1–9-2
  picking, 9-2–9-14
  selecting, 9-14–9-21
picksize, 9-9
PIECECHANGE, 7-15
pixels, writing and reading, 3-19–3-21
  readpixels, 3-20–3-21
  readRGB, 3-21
  writepixels, 3-19
  writeRGB, 3-20
pmv, 3-11; 10-6
pnt, 3-3; 10-6
points, 3-3–3-5
  example programs, 3-4–3-5
  pnt, 3-3
polarview, 4-7; (fig), 4-8–4-9
polf, 3-11
polling a device, 7-6–7-7
  getbutton, 7-6
  getdev, 7-7
  getvaluator, 7-6
polling and queuing, 7-1–7-4
  input buttons, 7-3
  input valuators, 7-3
  other devices, 7-4
  overview, 7-1–7-2
poly, 3-10
polygon removal, backfacing, see
  backfacing polygon removal
polygons, 3-10–3-13
  filled and unfilled (fig), 3-9
  pclos, 3-13
  pdr, 3-12
  pmv, 3-11
  polf, 3-11
  poly. 3-10
  rpdr, 3-12
  rpmv, 3-12
pop, W-49; W-50

popattach, W-49
popattributes, 2-7
popmatrix, 4-22
popname, 9-7
pop-up menu
  binding colors to, W-53
  calling up, W-34–W-36
  colors, choosing for, W-36–W-38
  colors for drawing, W-37
  creating, W-32–W-34
  defined, W-2
  example program, W-43–W-46; from *cedit*, W-41–W-43
  making, W-31–W-46
  menu formats, advanced, W-38–W-41
pop-up menu, calling up, W-34–W-36
  dopup, W-35–W-36
  freepup, W-36
pop-up menu, choosing colors for, W-36–W-38
  colors for drawing pop-up menus, W-37
  pupcolor, W-37
pop-up menu, creating, W-32–W-34
  addtppup, W-33–W-34
  defpup, W-32–W-33
  endpupmode, W-34
  newpup, W-32
  pupmode, W-34
pop-up menu routines, W-31
popviewport, 4-20
prefposition, W-12; W-14
prefsize, W-12; W-13
programming hints, window manager, W-24–W-25
  event queue, W-25
  graphics initialization, W-24
  raster fonts, W-24
  shared facilities, W-24
programming languages and Graphics Library, 1-5; see also C, FORTRAN, Pascal
programs, examples, see C, example programs and FORTRAN, example programs
projection transformations, see transformations, projection
pruning, 8-7
pupmode, W-31; W-34
push, W-49
pushattributes, 2-7
pushmatrix, 4-22
pushname, 9-7
pushviewport, 4-20

Q

qdevice, 7-7
qenter, 7-9
qread, 7-10
qreset, 7-11
qtest, 7-10
queuing, see polling and queuing

R

raster fonts, W-24
rational curves, 11-21–11-22
rcrv, 11-22
rcrvn, 11-22
rdr, 3-7
readpixels, 3-20–3-21
readRGB, 3-21
rect, 3-8
rectangles, 3-8–3-10
  filled and unfilled (fig), 3-9
  rect, 3-8
  rectcopy, 3-10
  rectf, 3-8
rectcopy, 3-10
rectf, 3-8
REDRAW, 7-15; W-57–W-64; example program in C, W-59–W-64
relative drawing, see drawing, relative
reservebut, W-48
reset linestyle, initial value, 2-2
resetls, 5-4
reshapeviewport, W-20; W-22
return values, changing for menu selections, W-39
RGBcolor, 6-14
RGB color, initial value, 2-2
RGBcursor, 6-23
RGBmode, 6-3
RGB mode, 6-3
RGBwritemask, 6-19
RGB writemask, initial value, 2-2
ringbell, 7-13
rmv, 3-7

rot, 4-3
rotate, 4-3
rpatch, 11-25
rpdr, 3-12
rpmv, 3-12

S

saving global state attributes, 2-7
  popattributes, 2-7
  pushattributes, 2-7
scale, 4-5
screen coordinate system, 4-1
screenspace, W-20; W-21
scrmask, 4-18
select, sample application (fig), 9-16
selecting, 9-14–9-21
  endselect, 9-15
  examples, 9-17–9-21
  gselect, 9-15
  overview, 9-14
  sample application (fig), 9-16
setbell, 7-13
setcursor, 6-20
setdblight, 7-14
setdepth, 12-3
setfastcom, 2-5
setlinestyle, 5-2
setmap, 6-10
setmonitor, 2-6
setpattern, 5-7
setshade, 13-4
setslowcom, 2-5
setvaluator, 7-5
shaded polygon, example program
  in C, 13-4–13-5; example program
  in FORTRAN, 13-5–13-6
shaderange, 13-7–13-8; initial value, 2-2
shading, 13-1–13-6
  example, 13-4–13-6
  getshade, 13-4
  Gouraud shading (fig), 13-3
  setshade, 13-4
  spclos, 13-4
  splf, 13-2
shared facilities, W-24
singlebuffer, 6-4
single buffer and double buffer
  modes, 6-3–6-7
  backbuffer, 6-5
  doublebuffer, 6-4
  finish, 6-7
  frontbuffer, 6-6
  getbuffer, 6-6
  getdisplaymode, 6-6
  getplanes, 6-7
  gsync, 6-7
  overview, 6-3–6-4
  singlebuffer, 6-4
  swapbuffers, 6-5
  swapinterval, 6-5
single buffer mode, example
  program, W-25–W-26
spclos, 13-4
special devices, 7-14–7-15
  cursor devices, 7-15
  ghost devices, 7-15
  keyboard devices, 7-14
  timer devices, 7-14
  window manager devices, 7-15
splf, 13-2
square drawn with points, example program
  in C, 3-4; example program
  in FORTRAN, 3-4–3-5
stepunit, W-12; W-14
strwidth, 5-12
surface patch, 11-23
  example program in C, 11-28–11-30
  example program in
    FORTRAN, 11-30–11-32
  surface patches (fig), 11-26–11-27
surfaces, see curves and surfaces
surfaces, drawing, 11-23–11-33
  example, 11-28–11-32
  joined patches (fig), 11-33
  overview, 11-23–11-24
  patch, 11-25
  patchbasis, 11-24
  patchcurves, 11-24
  patchprecision, 11-25
  rpatch, 11-25
  surface patches (fig), 11-26–11-27
surfaces, hidden, 12-1–12-13
  backfacing polygon removal, 12-6–12-13
  z-buffer mode, 12-1–12-6
swapbuffers, 6-5
swapinterval, 6-5

T

text, 3-16–3-19
  charstr, 3-17
  cmov, 3-16–3-17
  example program in C, 3-18;
  example program in
    FORTRAN, 3-18–3-19
textcolor, 14-2
textinit, 14-3
textport, 14-2
textports, 14-1–14-3
  gettp, 14-2
  pagecolor, 14-3
  pagewritemask, 14-3
  textcolor, 14-2
  textinit, 14-3
  textport, 14-2
  textwritemask, 14-3
  tpoff, 14-1
  tpon, 14-1
textwritemask, 14-3
tie, 7-9
timer devices, 7-14
title bar, W-2
  binding colors to, W-51–W-52
  menu (fig), W-4
  making, W-39
  title bar menu (fig), W-4
  title bar and border regions of
    a window (fig), W-52
title font, selecting, W-7
tpoff, 14-1
tpon, 14-1
transformations, coordinate, 4-1–4-23
  modeling transformations, 4-2–4-6
  projection transformations, 4-10–4-15
  systems overview, 4-1–4-2
  user-defined transformations, 4-20–4-23
  viewing transformations, 4-7–4-10
  viewports, 4-15–4-20
transformations, modeling, 4-2–4-6
  modeling routine order (fig), 4-6
  modeling routines (fig), 4-4
  rot, 4-3
  rotate, 4-3
  scale, 4-5
  translate, 4-5
transformations, projection, 4-10–4-15
  ortho, 4-15
  ortho (fig), 4-16
  ortho2, 4-15
  perspective, 4-12
  perspective (fig), 4-13
  window, 4-12
  window (fig), 4-14
transformations, user-defined, 4-20–4-23
  getmatrix, 4-23
  loadmatrix, 4-21
  multmatrix, 4-22
  overview, 4-20–4-21
  popmatrix, 4-22
  pushmatrix, 4-22
transformations, viewing, 4-7–4-10
  lookat, 4-10; (fig), 4-11
  polarview, 4-7; (fig), 4-8–4-9
translate, 4-5

U

unqdevice, 7-8
user-defined transformations, see
  transformations, user-defined
using objects, 8-5–8-8
  bbox2, 8-7–8-8
  bounding boxes (fig), 8-9
  callobj, 8-5–8-7
  hierarchical objects (fig), 8-6

viewing transformations, see
  transformations, viewing
viewport, 4-17
  clearing, 3-3
  initial value, 2-2
viewports, 4-15–4-20
  getscrmask, 4-18
  getviewport, 4-17
  popviewport, 4-20
  pushviewport, 4-20
  scrmask, 4-18
  viewport, 4-17

W

winat, W-20; W-24
winattach, W-17; W-19
winclose, W-11
winconstraints, W-16
window, 4-12; (fig), 4-14; W-1—2
window constraints, setting, W-11–W-17
  foreground, W-15–W-16
  fudge, W-14
  imakebackground, W-15
  keepaspect, W-13
  maxsize, W-12
  minsize, W-12
  noport, W-15
  overview, W-11–W-12
  prefposition, W-14
  prefsize, W-13
  routines, W-12
  setting for existing windows, W-16–W-17
  stepunit, W-14
  winconstraints, W-16–W-17
window control routines, W-17
window, graphics, W-9
window manager
  changing, see customizing *mex*
  default configuration, see *mex*,
    getting started with
  devices, 7-15
  programming hints, W-24–W-27
  terminology, W-2
window routines, other, W-19–W-24
  endfullscrn, W-22
  fullscrn, W-22
  getorigin, W-21
  getsize, W-21
  ismex, W-23
  miscellaneous, W-20
  reshapeviewport, W-22
  screenspace, W-21
  winat, W-24
  winget, W-23
  winset, W-23
  wintitle, W-20
windows
  attaching to, W-5
  changing noninteractively, W-17–W-19
  constraints, setting, W-11–W-17
  creating, W-3–W-4
  moving, W-5
  opening and closing, W-9–W-11
  other routines, W-19–W-24
  popping, W-6
  pushing, W-6
  removing, W-6
  reshaping, W-6
  selecting, W-5
windows, changing
  noninteractively, W-17–W-19
  winattach, W-19
  winmove, W-18
  winpop, W-19
  winposition, W-18
  winpush, W-19
windows, multiple, controlling from a
  single process, W-57–W-64
  creating windows, W-58
  drawing windows, W-58
  example, W-59–W-64
  processing of REDRAW token and
    gid, W-57
window, text, W-9
winget, W-20; W-23
winmove, W-17; W-18
winopen, W-9–W-10
winpop, W-17; W-19
winposition, W-17; W-18
winpush, W-17; W-19
winset, W-20; W-23
wintitle, W-20
world coordinate system, 4-1
writemask, 6-16; (fig), 6-17
writemasks, 6-16–6-19
  and color map (fig), 6-18
  getwritemask, 6-16
  gRGBmask, 6-19
  RGBwritemask, 6-19
  writemask, 6-16; (fig), 6-17; initial
    value, 2-2
writemasks and the color map (fig), 6-18
writepixels, 3-19
writeRGB, 3-20

X

xfpt, 10-6; 10-12–10-13; example program
  in C, 10-13–10-14; example program
  in FORTRAN, 10-14–10-16

Z

z-buffer mode, 12-1–12-6
  example, 12-4–12-6
  getdepth, 12-3
  getzbuffer, 12-4
  initial value, 2-2
  overview, 12-1
  setdepth, 12-3
  z-buffer, 12-4
  z-buffer bitplanes (fig), 12-2
  zclear, 12-3
z-buffer, 12-4; example program
  in C, 12-4–12-5; example program
  in FORTRAN, 12-5–12-6
z-buffer bitplanes (fig), 12-2
zclear, 12-3