The IRIS is a powerful tool for interactive computer graphics. It is ideal for a variety of graphics applications, including CAD/CAM, simulation, VLSI design, and document preparation. The IRIS Graphics Library is the set of graphics commands that provides high- and low-level support for graphics on the IRIS. The IRIS User's Guide describes the Graphics Library and consists of four major sections.

I. The Programming Guide discusses the Graphics Library commands in a narrative style. The first chapter is an IRIS System Overview and the succeeding chapters are a topically organized discussion of the commands.

II. The Window Manager section describes Multiple Exposure, the window manager for the IRIS workstation.

III. The Programming Examples section contains ten programs which illustrate the use of various Graphics Library commands.


The glossary defines graphics and IRIS-related terminology and the index identifies where key terms are discussed in the Programming Guide.

Finally, we welcome your comments and suggestions on the IRIS documentation. These will be a valuable source of ideas for improvement of the IRIS User's Guide. For this purpose, a postage-paid Reader Comment form is provided at the end of the guide.
TABLE OF CONTENTS

I. PROGRAMMING GUIDE

1. Introduction
   1.1 IRIS System Overview
   1.2 The Graphics Library
   1.3 Documentation Conventions

2. Global State Commands
   2.1 Initialization
   2.2 Saving Global State

3. Drawing Primitives
   3.1 Current Drawing Positions
   3.2 Clearing the Viewport
   3.3 Points
   3.4 Lines
   3.5 Rectangles
   3.6 Polygons
   3.7 Circles and Arcs
   3.8 Text
   3.9 Writing and Reading Pixels

4. Coordinate Transformations
   4.1 Modeling Transformations
   4.2 Viewing Transformations
   4.3 Projection Transformations
   4.4 Viewports
   4.5 User-Defined Transformations

5. Textures and Fonts
   5.1 Linestyles
   5.2 Patterns
   5.3 Fonts

6. Display Modes and Color
   6.1 Display Modes
   6.2 Colors Maps
   6.3 Colors and Writemasks
   6.4 Cursors
7. Input/Output Commands
   7.1 Initializing a Device
   7.2 Polling a Device
   7.3 The Event Queue
   7.4 Controlling Peripheral Input/Output Devices

8. Graphical Objects
   8.1 Defining an Object
   8.2 Using Objects
   8.3 Object Editing

9. Picking and Selecting
   9.1 Mapping the Screen to World Space
   9.2 Picking
   9.3 Selecting

10. Geometry Pipeline Feedback
    10.1 The Geometry Pipeline
    10.2 Feedback Mode

11. Curves and Surfaces
    11.1 Curve Mathematics
    11.2 Drawing Curves
    11.3 Drawing Surfaces

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

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

14. Textports

II. APPENDICES

Appendix A: Type Definitions for C and Pascal
Appendix B: Geometry Engine Computations
Appendix C: Transformation Matrices
Appendix D: Feedback Parser
III. INDEX

IV. MULTIPLE EXPOSURE: THE IRIS WINDOW MANAGER

1. Introduction
2. The Default User Interface to the Window Manager
3. Programming the Window Manager
4. Programming Examples
5. Programming Considerations
6. Customizing the User Interface

V. PROGRAMMING EXAMPLES

Example 1: Getting Started
Example 2: Common Drawing Commands
Example 3: Object Coordinate Systems
Example 4: Double Buffer Display Mode
Example 5: Input/Output Devices
Example 6: Interactive Drawing
Example 7: Popup Menus
Example 8: Modeling Transformations
Example 9: Writemasks and Color Maps
Example 10: A Color Editor

VI. IRIS REFERENCE MANUAL

VII. GLOSSARY
1. Introduction

The IRIS (Integrated Raster Imaging System) is a high-performance, high-resolution color computing system for 2D and 3D 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 IRIS is a custom VLSI chip called the Geometry Engine™. A pipeline of ten or twelve Geometry Engines accepts points, vectors, polygons, characters, and curves in user-defined coordinate systems and transforms them to screen coordinates, with rotation, clipping, and scaling. In addition to the Geometry Pipeline, an IRIS system consists of: a general purpose microprocessor, a raster subsystem, a high-resolution color monitor, a keyboard, and graphics input devices.

1.1 IRIS System Overview

Conceptually, the graphics hardware of the IRIS is divided into three pipelined components, as 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 commands issued by the applications program either are sent immediately through the pipeline, or are compiled into graphical objects (display lists of graphics commands) which can be called later.

Graphics commands are expressed in 2D or 3D user-defined coordinates. These commands are sent 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 coordinates. The output of the Geometry Pipeline is then sent to the raster subsystem. The raster subsystem fills in the pixels between the endpoints of the lines, fills in the interiors of polygons, converts character codes into bit-mapped characters, and performs shading, depth-cueing, and hidden surface removal. A color value for each pixel is stored in the system's bitplanes. The values contained in the bitplanes are then used to display an image on the monitor.

1.2 The Graphics Library

The IRIS Graphics Library is a set of graphics and utility commands that provide high- and low-level support for graphics on the IRIS. The IRIS system software is written in C, but the commands in the Graphics Library are callable in C, FOR-

Geometry Engine is trademark of Silicon Graphics, Inc.
Figure 1.1
The IRIS consists of three subsystems: the Applications/Graphics Processor, the Geometry Pipeline, and the Raster Subsystem.
TRAN, and Pascal. The graphics commands can be broken into the following categories:

- **Global state** commands initialize the hardware and control global state variables.
- **Primitive drawing** commands draw points, lines, polygons, circles, arcs, and text strings on the screen.
- **Drawing attribute** commands select characteristics for drawing lines, filling polygons, and writing text strings.
- **Coordinate transformation** commands perform manipulations on coordinate systems, including mapping user-defined coordinate systems to screen coordinate systems.
- **Display mode and color** commands determine how the bitplane image memory is used and how objects are colored on the screen.
- **Input/output** commands initialize and read input/output devices.
- **Object creation and editing** commands provide the means to create hierarchical structures of graphics commands.
- **Picking and selecting** commands identify the commands that draw to a specified area of the screen.
- **Geometry Pipeline feedback** commands provide access to the computing capabilities of the geometry hardware.
- **Curve and surface** commands draw complex curved lines and surfaces.
- **Hidden surface** commands activate 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.
- **Shading** commands draw Gouraud-shaded polygons.
- **Depth-cue** commands draw points, lines, curves, and surfaces with intensities that vary as a function of depth.
- **Textport** commands allocate an area of the screen for writing text.

Each of the above categories is discussed in a separate chapter of the Programming Guide. Additional material is covered in the appendices:

- Appendix A contains header files for IRIS 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 commands in the Graphics Library.
- Appendix D discusses a feedback parser which simplifies the use of the Geometry Engines in feedback mode.
1.3 Documentation Conventions

The IRIS Graphics Library is accessible from three programming languages: C, FORTRAN, and Pascal. Specifications for the commands in the library are separated from the text by horizontal lines. The C specification is given first, the FORTRAN second, and the Pascal last. For example,

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

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

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

Each command has a root name, like move. The default world space is 3D with floating-point coordinates. Suffixes are added to some commands 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)`
- `movei(1, 2, 3)`
- `moves(1, 2, 3)`
- `move2(1.0, 2.0)`
- `move2i(1, 2)`
- `move2s(1, 2)`

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

While FORTRAN programs are constrained to a small set of predefined data types, both 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 XMAX-SCREEN. 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 programs are written in C, with the objective of making the programs clear rather than efficient. FORTRAN versions of the following programs have also been included in the text: the picking example in Chapter 9, the curve and the surface patch examples in Chapter 11, and examples #2 and #7 in the Programming Examples section.
2. Global State Commands

This chapter tells how to initialize an IRIS program, exit an IRIS program, and control the global state variables.

2.1 Initialization

The first Graphics Library command in every IRIS program is \texttt{ginit} or \texttt{gbegin}. These commands initialize the hardware, allocate memory for symbol tables and display list objects, and set up default values for global state variables. They have no arguments and should be called only once (before any other library command).

\begin{verbatim}

ginit()

subroutine ginit

procedure ginit;
\end{verbatim}

\texttt{gbegin} performs all the same tasks as \texttt{ginit}, except it does not alter the color map.

\begin{verbatim}

gbegin()

subroutine gbegin

procedure gbegin;
\end{verbatim}
<table>
<thead>
<tr>
<th>State Variable</th>
<th>Initial Value</th>
<th>Section #</th>
</tr>
</thead>
<tbody>
<tr>
<td>available bitplanes</td>
<td>all bitplanes(^{(1)})</td>
<td>6.1</td>
</tr>
<tr>
<td>backface mode</td>
<td>off</td>
<td>12.2</td>
</tr>
<tr>
<td>color</td>
<td>undefined</td>
<td>6.3</td>
</tr>
<tr>
<td>color map mode</td>
<td>one map</td>
<td>6.2</td>
</tr>
<tr>
<td>cursor</td>
<td>0 (arrow)(^{(2)})</td>
<td>6.4</td>
</tr>
<tr>
<td>depthcue mode</td>
<td>off</td>
<td>13.2</td>
</tr>
<tr>
<td>display mode</td>
<td>single buffer</td>
<td>6.1</td>
</tr>
<tr>
<td>font</td>
<td>0 (^{(3)})</td>
<td>5.3</td>
</tr>
<tr>
<td>linestyle</td>
<td>0 (solid)</td>
<td>5.1</td>
</tr>
<tr>
<td>linestyle backup</td>
<td>off</td>
<td>5.1</td>
</tr>
<tr>
<td>linestyle repeat</td>
<td>1</td>
<td>5.1</td>
</tr>
<tr>
<td>linewidth</td>
<td>1 pixel</td>
<td>5.1</td>
</tr>
<tr>
<td>monitor type</td>
<td>monitor supplied(^{(4)})</td>
<td>2.1</td>
</tr>
<tr>
<td>pattern</td>
<td>0 (solid)</td>
<td>5.2</td>
</tr>
<tr>
<td>picking size</td>
<td>10 x 10 pixels</td>
<td>9.2</td>
</tr>
<tr>
<td>reset linestyle</td>
<td>on</td>
<td>5.1</td>
</tr>
<tr>
<td>RGB color</td>
<td>undefined</td>
<td>6.3</td>
</tr>
<tr>
<td>RGB writemask</td>
<td>undefined</td>
<td>6.3</td>
</tr>
<tr>
<td>shaderange</td>
<td>0,7,0,1023</td>
<td>13.2</td>
</tr>
<tr>
<td>viewport</td>
<td>entire screen</td>
<td>4.4</td>
</tr>
<tr>
<td>writemask</td>
<td>all planes enabled(^{(1)})</td>
<td>6.3</td>
</tr>
<tr>
<td>zbuffer mode</td>
<td>off</td>
<td>12.1</td>
</tr>
</tbody>
</table>

Table 2.1: Initial values of global state variables

(1) If there are more than three bitplane boards installed, available planes = twelve.
(2) The color and writemask of the cursor are set to 1.
(3) Rasterfont 0 is a Helvetica-like font.
(4) A PROM on the graphics processor tells the IRIS which monitor it has.
<table>
<thead>
<tr>
<th>Index</th>
<th>Name</th>
<th>RGB Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Red</td>
</tr>
<tr>
<td>0</td>
<td>BLACK</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>RED</td>
<td>255</td>
</tr>
<tr>
<td>2</td>
<td>GREEN</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>YELLOW</td>
<td>255</td>
</tr>
<tr>
<td>4</td>
<td>BLUE</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>MAGENTA</td>
<td>255</td>
</tr>
<tr>
<td>6</td>
<td>CYAN</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>WHITE</td>
<td>255</td>
</tr>
<tr>
<td>all others</td>
<td>unnamed</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: Initial color map values

greset returns the global state variables to their initial values, and can be called at any time. The global state variables are listed in Table 2.1. The color map is initialized by greset to the values shown in Table 2.2. greset also performs the following tasks (which are discussed in more detail throughout the Programming Guide):

- sets up a two-dimensional orthographic projection transformation that maps user-defined coordinates to the entire area of the screen (see Section 4.3);
- turns on the cursor and ties it to MOUSEX and MOUSEY (see Sections 6.4 and 7.3);
- unqueues each button, each valuator, and the keyboard (see Chapter 7);
- sets each button 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).

---

greset()

subroutine greset

procedure greset;

When the IRIS is used as a terminal, most graphics commands are buffered at
the host by the communication software for efficient block transfer of data from
the host to the IRIS. gflush causes all buffered yet untransmitted graphics data to
be delivered to the IRIS. Certain graphics commands (notably those which re-
turn values) cause the host buffer to be flushed when they are executed. On the
IRIS workstation, gflush does nothing.

\begin{verbatim}
gflush()

subroutine gflush

procedure gflush;
\end{verbatim}

The final graphics command in an IRIS program is gexit. gexit flushes commu-
nication buffers and waits for the graphics pipeline to empty.

\begin{verbatim}
gexit()

subroutine gexit

procedure gexit;
\end{verbatim}

The setmonitor command selects one of three types of monitors: HZ30 = 30hz in-
terlaced, HZ60 = 60hz non-interlaced, and NTSC = NTSC (television standard
encoding).

\begin{verbatim}
setmonitor(type)
short type;

subroutine setmon(type)
integer*4 type

procedure setmonitor(type: Short);
\end{verbatim}

The getmonitor command returns the monitor type currently selected.
long getmonitor()

integer+4 function getmon()

function getmonitor: longint;

2.2 Saving Global State

The pushattributes command saves the current global state. The IRIS maintains a
stack of attributes, and 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, front and back buffers.
These attributes are discussed in Chapters 5 and 6.

pushattributes()

subroutine pushat

procedure pushattributes;

popattributes restores the most recently saved values of the global state variables.

popattributes()

subroutine popatt

procedure popattributes;
3. Drawing Primitives

The IRIS Graphics Library includes commands for drawing points, lines, rectangles, polygons, circles, arcs, curves, and surfaces. It also includes commands for shading surfaces, for drawing text strings, and for writing and reading pixels. These commands take positioning arguments that may be integers (24 bits), short integers (16 bits), or floating-point numbers. Points, lines, polygons, and text can be positioned in two or three dimensions. Rectangles, circles, and arcs are defined in 2D, but can be translated to 3D using the modeling commands described in Chapter 4. Curves are always specified in 3D, although the z coordinate can be zero. Surface patches are specified in 3D.

3.1 Current Drawing Positions

The IRIS maintains two current drawing positions which determine where drawing takes place when a drawing command is executed.

The current graphics position is specified in 3D with floating-point coordinates and is updated by all drawing commands, except text, pixel, and clear commands. The current graphics position has meaning during streams of move, draw, and pat commands. These commands can be interwoven with attribute-setting commands. (See Chapters 5 and 6 for a discussion of attributes.) Most other commands leave the graphics position undefined.

The current character position is specified in 2D with screen coordinates and has meaning during streams of cmov, charstr, and the four routines that read and write pixels. In general, only the attribute commands leave the character position intact.

The getgpos command returns the current graphics position, after transformation and before clipping and scaling. (The fourth coordinate, fw, is used 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);
```
The `getcpos` command returns the current character position in screen coordinates.

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

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

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

### 3.2 Clearing the Viewport

The `clear` command sets the screen area within the current viewport to the current color using the current writemask and texture. (See Chapter 4 for a discussion of viewports and Chapters 5 and 6 for descriptions of the color, writemask, and texture attributes.) `clear` leaves the current graphics position and the current character position undefined.

```plaintext
clear()

subroutine clear

procedure clear;
```

### 3.3 Points

The `pt` command draws a point. If the point is visible on the screen, it is shown as one pixel using the current color. `pt` updates the current graphics position to its specified 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 screen:
#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();
}

3.4 Lines
The move and draw commands draw lines. The move command sets the current graphics position to the point specified by its arguments.
move(x, y, z)
Coord x, y, z;

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

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

The **draw** command draws a line from the current graphics position to the point specified by its arguments. The appearance of the line is determined by the current linestyle, linewidth, linestyle repeat, linestyle backup, color, and writemask (see Chapters 5 and 6). **draw** updates the current graphics position to the point specified by its arguments.

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 a blue box on the screen using **move** and **draw** commands. For simplicity, a 2D integer world space is assumed.

```
#include "gl.h"

main()
{
    ginit();
curseoff();
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 */
```
Relative drawing

The relative drawing commands interpret their arguments using the current graphics position as an origin. For example, the command

\[
\text{rmv}(a,b,c)
\]

changes the current graphics position from \((x,y,z)\) to \((x+a,y+b,z+c)\). The command

\[
\text{rdr}(a,b,c)
\]

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 \((x+a,y+b,z+c)\).

---

\[
\text{rmv}(dx, dy, dz)
\]
Coord \(dx, dy, dz;\)

subroutine \text{rmv}(dx, dy, dz)
real \(dx, dy, dz\)

procedure \text{rmv}(dx, dy, dz: Coord);

---

\[
\text{rdr}(dx, dy, dz)
\]
Coord \(dx, dy, dz;\)

subroutine \text{rdr}(dx, dy, dz)
real \(dx, dy, dz\)

procedure \text{rdr}(dx, dy, dz: Coord);

3.5 Rectangles

A rectangle is determined by two points specifying opposite corners. The sides of the rectangle are parallel to the x and y axes; the z coordinate is zero. \text{rect} outlines a rectangle and \text{rectf} draws a filled rectangle. Since a rectangle is a two-dimensional shape, these commands take only 2D arguments, and set z to zero.
rect takes four arguments: $x_1$, $y_1$, $x_2$, and $y_2$. As shown in Figure 3.1, a rectangle is outlined by four line segments using the current linestyle, linewidth, linestyle repeat, linestyle backup, color, and writemask.

```
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 takes the same arguments as rect and produces a filled rectangular region. The currently selected texture pattern, color, and writemask are used. An example is shown in Figure 3.1.

Both rect and rectf set the current graphics position to $(x_1, y_1)$.

```
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);
```

The rectcopy command copies a rectangular array of pixels defined in screen coordinates 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; polif fills a polygonal area. A polygon is represented by an array of points. The first and last points are automatically connected to close the polygon.

poly takes two arguments: the number of points in the polygon and an array of coordinates. Points can be expressed in 2D or 3D, using integers, shorts, or floating-point numbers. The polygon is drawn using the current linestyle, linestyle repeat, linestyle backup, linewidth, color, and writemask.

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

```plaintext
subroutine poly(n, array)
integer*4 n
real array(3,n)

procedure poly(n: longint; var array: Coord3array);
```

polif takes the same arguments as poly, but fills the polygon using the current texture pattern, color, and writemask. All filled polygons must be convex. Although no errors are reported if concave polygons are specified, they produce unpredictable results.

```plaintext
polif(n, array)
long n;
Coord array[][3];
```

```plaintext
subroutine polif(n, array)
integer*4 n
real array(3,n)

procedure polif(n: longint; var array: Coord3array);
```

Figure 3.1 shows a polygon drawn with both poly and polif. A solid linestyle and solid texture pattern are shown.

poly and polif set the current graphics position to the first point in array.

Filled polygons can also be drawn by specifying one vertex at a time. The pmv command specifies the first point in a polygon. A sequence of pdr commands
Figure 3.1
Rectangles are specified by opposite corners. A polygon is defined by an array of points. Both rectangles and polygons can be drawn as outlined or filled areas.
then draws lines to each of the subsequent points in the polygon (using the current linestyle, linewidth, linestyle repeat, linestyle backup, color, and writemask). The \texttt{pclos} command connects the last point with the first. The results of the \texttt{pmv}, \texttt{pdr}, and \texttt{pclos} commands are undefined if the polygon is not convex.

\begin{verbatim}
pmv(x, y, z)
Coord x, y, z;

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

procedure pmv(x, y, z: Coord);
\end{verbatim}

\begin{verbatim}
pdr(x, y, z)
Coord x, y, z;

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

procedure pdr(x, y, z: Coord);
\end{verbatim}

The \texttt{rpmv} command specifies the first point in a polygon using the current graphics position as an origin. The \texttt{rpdr} command draws a line to a point in the polygon using the previous point (the current graphics position) as an origin.

\begin{verbatim}
rpmv(dx, dy, dz)
Coord dx, dy, dz;

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

procedure rpmv(dx, dy, dz: Coord);
\end{verbatim}
rpdr(dx, dy, dz)
Coord dx, dy, dz;

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

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

After a sequence of pdrw or rpdr commands, the pclose command connects the last point in the polygon with the first.

pclose()

subroutine pclose

procedure pclose;

3.7 Circles and Arcs

circ and circf draw outlined and filled circles. A circle is defined by a center point and a radius, in the x-y plane, with z = 0. Circles are drawn 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);

Filled circles are drawn with circf. The current color, texture pattern, and writemask are used to fill the circle with center \((x,y)\) and radius \(radius\). circ and circf set the current graphics position to \((x+radius, y)\).
Figure 3.2
Circular arcs are defined by a center point, radius, start angle, and end angle. They are drawn counterclockwise, with angles measured from the x-axis.
circf(x, y, radius)
Coord x, y, radius;

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

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

arc and arcf draw circular arcs. Arcs are defined by a center point, a radius, a starting angle, and an ending angle. The angles are measured from the x axis and are specified in integral tenths of degrees; positive angles describe counterclockwise rotations. The arc is drawn 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(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 produces filled arcs. The current texture pattern, color, and writemask are used. An example is shown in 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, startang, endang)
real x, y, radius
integer*4 startang, endang

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

3.8 Text
The current character position (see Section 3.1) determines where text is drawn on the screen. The cmov command sets the current character position. Its arguments specify a point in 2D or 3D, which is transformed into screen coordinates and becomes the new character 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 draws a string of characters. The origin of the first character in the string is placed at the current character position. After the string is drawn, the current character position is updated 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 currently selected font and color. (See Section 5.3 for a discussion of fonts.)
Gross clipping removes all strings that start outside the viewport. Fine clipping trims individual characters to the screenmask.
charstr(str)
String str;

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

procedure charstr(str: pstring);

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, however, the characters will be individually clipped to the screenmask. (Viewports and screenmasks are discussed in the next chapter.) Although the screenmask is normally set equal to the viewport, it can be set smaller than the viewport to enable two kinds of clipping. As shown in Figure 3.3, gross clipping removes all strings that start outside the viewport. Fine clipping trims individual characters to the screenmask.

The following example draws two lines of text. The program assumes the currently selected font is less than twelve pixels high.

```c
#include "gl.h"

main()
{
    glinit();
    cursoff(); /* turn the cursor off so it won't interfere
    with the text */
    color(BLACK);
    clear();
    color(RED);
    cmove2i(300,380);
    charstr("The first line is drawn ");
    charstr("in two parts. ");
    cmove2i(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();
}
3.9 Writing and Reading Pixels

The `writepixels` and `writeRGB` commands paint one or more pixels on the screen. The commands specify the number of pixels to paint and a color for each pixel. The current character position is the starting location; it is updated to the pixel that follows the last one painted. The current character position becomes undefined if the next pixel position is greater than XMAXSCREEN. Pixels are painted from left to right and are clipped to the current screenmask (see Chapter 4). These commands do not automatically wrap from one line to the next.

`writepixels` supplies the number of pixels to be painted and a color index for each pixel. (See Chapter 6 for an explanation of color maps.)

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

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

procedure writepixels(n: Short; var colors: Colorarray);
```

`writeRGB` supplies a 24-bit RGB value (eight bits each for red, green, and blue) for each pixel. It is written directly into the bitplanes. (See Chapter 6 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: Short; var red, green, blue: RGBarray);
```

It is also possible to read pixel values from image memory. `readpixels` attempts to read up to a 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 actually read. The values of pixels read outside the
current viewport are undefined. The current character position is updated to the
pixel to the right of the last one read, or is undefined if the new position is out-
side the viewport. `readpixels` will not wrap to the next line of pixels when the
edge of the screen is encountered.

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

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

function readpixels(n: Short; var colors: Colorarray): longint;
```

`readRGB` attempts to read up to `n` pixel values from the bitplanes. 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` re-
turns the number of pixels actually read. The values of pixels read outside the
current viewport are undefined. The current character position is updated to the
pixel to the right of the last one read, or is undefined if the new position is out-
side the viewport.

```c
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: Short; var red, green, blue: RGBarray): longint;
```
4. Coordinate Transformations

Chapter 3 discussed commands for drawing points, lines, circles, arcs, rectangles, and polygons. These commands are the primitives for drawing images. However, they can also be thought of as primitives for building models. For example, a sequence of drawing commands can be used to define a 3-dimensional shape. Such a sequence of commands is called a graphical object, and can be used to define a geometric model. Graphical objects can be manipulated as single entities – they can be moved, scaled, rotated, or combined with other objects into more complex objects. The coordinate transformation commands discussed in this chapter perform those manipulations. In addition, they map objects onto the screen so they can be displayed.

The process of displaying images of complex models can be broken into two sub-processes:

- building and manipulating models, and
- projecting models onto the screen.

A model is built from graphical objects. It is convenient to define each object in terms of its own origin. For example, a planet might be drawn using the center of the planet as the origin for each drawing command in the object. An object is said to be defined in object space when a convenient point is chosen as the object's origin, and component parts are placed relative to it. Several objects may be combined to form a composite object by mapping their individual coordinate systems to a coordinate system based on a new origin. The new composite object and its subobjects are defined in the new object space.

When a group of objects is to be displayed, it is defined in terms of one final object coordinate system, which is then called world space. In order to display the "world", a point of view must be specified. This point of view becomes the origin for a third coordinate system called eye space, and determines how the "world" is viewed. All the objects to be displayed are then defined in terms of the new coordinate system with origin at the view point.

Finally, the eye-space coordinate system must be mapped to a two-dimensional coordinate system that can be projected on the screen. This coordinate system is called screen space, and its origin is the lower left corner of the image.

The coordinate transformation commands perform the mappings from object space to world space to eye space and finally to screen space. There are four types of transformation commands:

- **Modeling transformation commands** transform the coordinate systems of objects. These commands include rotate, translate, and scale.

- **Viewing transformation commands** specify a viewing position, and
Figure 4.1
The modeling commands are *rotate*, *translate*, and *scale*. The object shown in (a) is rotated in (b), translated in (c), and scaled in (d) and (e).
transform a world coordinate system to an eye coordinate system. These commands include polarview and lookat.

- Projection transformation commands transform an eye coordinate system to a screen coordinate system. These commands include perspective, window, ortho, and ortho2.

- Viewport commands define a rectangular region of the screen for displaying an image (one screen coordinate system is transformed into another). These commands include viewport and scrmask.

After a coordinate transformation command is executed, all subsequent commands are interpreted through the new frame of reference. For example, if a translate command is executed, subsequent commands act on a translated coordinate system. If the perspective command has been executed, all subsequent objects will be projected on the screen with the specified perspective.

The coordinate transformation commands should be issued in the following order: a projection transformation command, a viewing transformation command, and finally any number of modeling commands. Each coordinate transformation is incorporated into the current transformation matrix, which reflects the cumulative effect of all the transformations. The coordinates of every drawing command are multiplied by the current transformation matrix. The current transformation matrix is the top matrix in a stack of eight 4x4 floating-point matrices. (The matrix stack can be extended in software to thirty-two matrices.) There are five commands for manipulating 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 (object space). The entire object can be manipulated using the modeling transformation commands: rotate, translate, and scale. By combining or concatenating these primitives, more complex modeling transformations can be generated to express relationships between different parts of a complex object.

The rotate command specifies an angle and an axis of rotation. The angle is given in tenths of degrees according to the right-hand rule – as you look down the positive rotation axis to the origin, positive rotation is counterclockwise. The axis of rotation is defined by a character, either 'x', 'y', or 'z' (the character can be upper or lower case). 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 the rotate command is executed are rotated. pushmatrix and popmatrix, described later in this chapter, preserve and restore an unrotated world space.
rotate(a, axis)
Angle a;
char axis;

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

procedure rotate(a: Angle; axis: char);

The **translate** command places the object space origin at a given world coordinate point. The object in Figure 4.1(a) is translated in Figure 4.1(c). All objects drawn after execution of the **translate** command will be translated. Again, **pushmatrix** and **popmatrix** may be used to limit the scope of the translation.

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

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

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

The **scale** command shrinks, expands, and mirrors objects. Its three arguments 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 drawn after the **scale** command is executed are scaled; **pushmatrix** and **popmatrix** can be used to limit the scope of **scale**. The object shown in Figure 4.1(a) is shrunk to one quarter of its original size and mirrored about the y axis in Figure 4.1(d). It is scaled only in the x direction in Figure 4.1(e).
Figure 4.2
The modeling commands are not commutative. Frame (a) shows the rotated and translated object; in (b), the object has been translated and rotated.
Figure 4.3

*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(0., 0., 0., 0.)

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

polarview(10., 0., 300., 0.)
scale(x, y, z)
float x, y, z;

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

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

rotate, translate, and scale can be combined to produce more complicated transformations. The order in which these transformations are applied is important. Figure 4.2 shows two different orderings of translate and rotate, with different results.

4.2 Viewing Transformations
The viewing transformations place the viewer and the eye coordinate system in world space. In the process, they define the world coordinate system. The polarview and lookat commands define a right-handed world coordinate system with x to the right, y up, and z toward the viewer. All rotations are specified 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 counter-clockwise. Users may choose other orientations for the world space, and can form viewing transformations from the modeling commands described in the next section.

polarview and lookat specify points and directions of view. The eye is placed at the point of view and looks at the world coordinate system's origin. A viewing transformation command should always follow a projection command.

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: Angle);

lookat specifies a viewpoint and a reference point on the line of sight in world coordinates. The viewpoint is at \((v_x, v_y, v_z)\) and the reference point at \((p_x, p_y, p_z)\). These two points define the line of sight. \texttt{twist} measures right-handed rotation about the z axis in the eye coordinate system. Figure 4.4 illustrates \texttt{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 twist

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

4.3 Projection Transformations

Projection transformations define the mapping from an eye-fixed coordinate system to the screen. The eye is placed in a right-handed system with the viewer at the origin looking down the negative z axis. Associated with each projection transformation is a viewport specifying a screen area to display the projected image.

The \texttt{perspective} and \texttt{window} commands specify perspective viewing pyramids into the world coordinate system and differ only in the method of defining the pyramid. The \texttt{ortho} command defines a 3D viewing rectangular parallelepiped and \texttt{ortho2} defines a 2D viewing rectangle for orthographic projections.

\texttt{perspective} defines the viewing pyramid by indicating the field-of-view angle \texttt{fovy} in the y direction of the eye coordinate system, the aspect ratio which determines the field of view in the x direction, and the location of the near and far
Figure 4.4

Lookat defines a viewpoint, a reference point along the line of sight, and a twist angle. The first pair of pictures shows the viewer and viewed image with no twist; twist is added to the second pair of frames.
Figure 4.5
The *perspective* command defines a field of view, an aspect ratio, and near and far clipping planes.
Figure 4.6
The window command defines a viewing window. A perspective view of the image is projected onto the window.
clipping planes in the z direction. The aspect ratio is given as a ratio of x to y. In general, the aspect ratio given in the \texttt{perspective} command should match the aspect ratio of the associated viewport. For example, \texttt{aspect = 2} means the viewer sees twice as far in x as in y. If the viewport is also twice as wide as it is tall, the image is displayed without distortion. The arguments \texttt{near} and \texttt{far} are distances from the viewer to the near and far clipping planes, and are always positive. Figure 4.5 illustrates the \texttt{perspective} command.

\begin{verbatim}
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: Angle; aspect: real; near, far: Coord);
\end{verbatim}

\texttt{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. The image is projected with perspective onto the screen. See Figure 4.6.

\begin{verbatim}
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);
\end{verbatim}

\texttt{ortho} defines a box-shaped enclosure in the eye coordinate system. \texttt{left}, \texttt{right}, \texttt{bottom}, and \texttt{top} define the x and y clipping planes. \texttt{near} and \texttt{far} are distances along the line of sight and can be negative. In other words, the z clipping planes are located at \(z = -\text{near}\) and \(z = -\text{far}\). Figure 4.7 shows an example of a 3D 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 defines a 2D clipping rectangle. When ortho2 is used with 3D world coordinates, the z values are left unchanged.

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 area of the screen where an image is displayed is called a viewport and is specified in screen coordinates. The visible screen area is 1024 pixels wide and 768 pixels high. The arguments to the viewport command define a rectangular area on the screen by the left, right, bottom, and top coordinates.

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);

The getviewport command returns the current viewport. The arguments to getviewport are the addresses of four memory locations. These will be assigned the left, right, bottom and top coordinates of the viewport.
Figure 4.7
The *ortho* command defines a viewing window. An orthographic view of the image is projected onto the window.
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);

A call to viewport sets both the viewport and the screenmask to the same area. A call to scrmask sets only the screenmask, which should be placed entirely within the viewport. Character strings that begin outside the viewport are clipped out; this is called gross clipping. Strings which begin inside the viewport but outside the screenmask are clipped to the screenmask. This is called fine clipping. For an illustration of character clipping, see Figure 3.3.

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 returns the screen 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);

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

subroutine pushvi

procedure pushviewport;

The popviewport command 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 is expressed as a 4x4 floating point matrix. Complex transformations can be built by concatenating a series of primitive ones. If $M$, $V$, and $P$ are modeling, viewing, and projection transformations, then the transformation $S$ that maps object space into screen space can be formulated as:

$$ S = M V P $$

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

The clipping boundaries are

$$ x = \pm w, \ y = \pm w, \text{ and } z = \pm w. $$

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 hardware stack that can hold up to eight transformation matrices. (The stack can be extended in software to thirty-two matrices.) The matrix on top of the stack, the current transformation matrix, is applied to all coordinate data.

The Geometry Pipeline forms a complex transformation matrix by pre-multiplying the current matrix by each primitive transformation. The transformation 5 defined above is formed by executing coordinate transformation commands in reverse order: first projection commands, then viewing commands, and finally modeling commands. Note that P is loaded onto the matrix stack, while both V and M pre-multiply the current matrix.

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

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(m: Matrix);
```

multmatrix pre-multiplies 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(m: Matrix);
```

pushmatrix pushes down the stack, duplicating the current matrix. If the stack contains one matrix, M, after a pushmatrix command it will contain two copies of M. Only the top copy can be modified.
pushmatrix()

subroutine pushma

procedure pushmatrix;

popmatrix pops the transformation stack.

popmatrix()

subroutine popmat

procedure popmatrix;

getmatrix copies the transformation matrix from the top of the stack to an array provided by the user; the stack is unchanged.

getmatrix(m)
Matrix m;

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

procedure getmatrix(var m: Matrix);
5. Textures and Fonts

Chapter 3 presented commands for drawing images. This chapter presents commands that control the characteristics, or attributes, of images when they are displayed on the screen. Attributes such as linestyles, texture, and font determine precisely which pixels are to be drawn when a drawing command is executed. For example, the linestyle determines whether a line appears as a solid line, or as a series of dashes, or as some other pattern.

5.1 Linestyles

Lines are drawn on the IRIS monitor using a 16-bit pattern called a linestyle. This pattern is used cyclically to determine which pixels in a 16-pixel line segment are to be colored. For example, the linestyle 0xFFFF draws a solid line, while 0xFOFO draws a dashed line and 0x8888 draws a dotted one. 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 non-solid lines.

The deflinestyle command defines a linestyle. The arguments specify an index into a table where linestyles are stored and a sixteen-bit linestyle pattern. There are \(2^{16}\) possible linestyle patterns and up to 128 of those patterns can be defined at one time. By default, index 0 contains the pattern 0xFFFF, which draws solid lines. The pattern at index 0 cannot be redefined.

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

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

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

There is always a current linestyle; it is used for drawing lines and curves and for outlining rectangles, polygons, circles, and arcs. The default linestyle is linestyle 0. Another pattern can be chosen using the setlinestyle command. Its argument is an index into the linestyle table built by calls to deflinestyle.
setlinestyle(index)
short index;

subroutine setlin(index)
integer*4 index

procedure setlinestyle(index: Short);

Four commands modify the application of the linestyle pattern. The first command, lsbackup, guarantees that a line will have a clearly marked endpoint. Normally, the current linestyle is used as a rotating pattern:

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

One implication of this algorithm is that the line may end without a clearly marked endpoint. If enabled, the lsbackup command guarantees the last two pixels in a line will be drawn. It takes one argument, a Boolean. If \( b = \text{TRUE} \), backup mode is enabled. FALSE, the default setting, means the linestyle will be used as is, and lines may have invisible endpoints.

lsbackup(b)
Boolean b;

subroutine lsback(b)
logical b

procedure lsbackup(b: Boolean);

Normally, a fresh copy of the linestyle is used for each new line. To draw a series of line segments with a continuous pattern, the resetls command is used. It is used most often for drawing circles, arcs, or curves, since curved lines are approximated with many short straight lines. If the linestyle is not reset between segments, the pattern of the curve appears smooth and continuous.

resetls has one Boolean argument. FALSE turns the mode off: the linestyle will not be reset between segments. TRUE, the default, means that each line will
start with a fresh copy of the pattern. Calls to \texttt{resetls} have the side effect of initializing the linestyle, no matter what the argument. If \texttt{resetls} is \texttt{FALSE} when \texttt{setlinestyle} is called, the linestyle is not changed until \texttt{resetls} is called. \texttt{resetls} should be set to \texttt{TRUE} when linestyle backup mode is enabled.

\begin{verbatim}
resetls(b)
Boolean b;

subroutine resetl(b)
logical b

procedure resetls(b: Boolean);
\end{verbatim}

The \texttt{lsrepeat} command is used to specify patterns longer than sixteen bits. The pattern in the current linestyle is multiplied by the integer argument to \texttt{lsrepeat}. Each 0 in the linestyle pattern becomes a series of \texttt{factor} 0's, and each 1 becomes a series of \texttt{factor} 1's.

\begin{verbatim}
lsrepeat(factor)
long factor;

subroutine lsrepe(factor)
integer*4 factor

procedure lsrepeat(factor: longint);
\end{verbatim}

The final command that modifies line drawing is \texttt{linewidth}, which specifies the width of a line. The width can be measured in pixels along the x axis, or along the y axis. The IRIS defines the width of a line to be the number of pixels along the axis which has the smallest change between the endpoints of the line. If \texttt{linewidth} is set to \( n > 1 \), \texttt{resetls} must be \texttt{TRUE}. 
Pattern solid

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

Pattern checked

= { 0x3333, 0x3333, 0xC CCC, 0xC CCC, 0x3333, 0x3333, 0xC CCC, 0xC CCC, 0x3333, 0x3333, 0xC CCC, 0xC CCC, 0x3333, 0x3333, 0xC CCC, 0xC CCC } 

Pattern halftone

= { 0x5555, 0xA AAA, 0x5555, 0xA AAA, 0x5555, 0xA AAA, 0x5555, 0xA AAA, 0x5555, 0xA AAA, 0x5555, 0xA AAA, 0x5555, 0xA AAA, 0x5555, 0xA AAA } 

Pattern crosshatch

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

Figure 5.1

A pattern is a 16 × 16, 32 × 32, or 64 × 64 array of bits with the origin in the lower left corner.
linwidth(n)
short n;

subroutine linewi(n)
integer*4 n

procedure linewidth(n: Short);

The current values of these line drawing attributes can be interrogated by the user.
getlstyle returns the index of the currently selected pattern.

long getlstyle()

integer*4 function getlst()

function getlstyle: longint;

getlsbackup returns the current value of the linestyle backup flag. Again, TRUE means that the last two pixels of a line will be colored regardless of the linestyle pattern. FALSE is the default.

long getlsbackup()

logical function getsb()

function getsbackup: longint;

getresetls returns the current value of the reset linestyle flag. TRUE means that the linestyle will be reinitialized for each line segment and is the default value. FALSE means that the pattern will rotate continuously across line segment boundaries.
Boolean getresetls()

logical function getres()

function getresetls: longint;

getlsrepeat returns the linestyle replication factor.

long getlsrepeat()

integer*4 function getlsr()

function getlsrepeat: longint;

Finally, the getlinewidth command returns the current linewidth in pixels.

long getlinewidth()

integer*4 function getlwi()

function getlinewidth: longint;

5.2 Patterns

Rectangles, polygons, and arcs can be filled with arbitrary textures. A texture pattern is an array of short integers that defines a rectangular pixel pattern. The texture pattern controls which pixels will be colored when filled objects are drawn. The pattern is aligned to the lower left corner of the screen rather than to the edge of the filled shape, so that it seems continuous over large areas.

Texture patterns are defined with the defpattern command. The arguments specify an index into a table of patterns, a size, and an array of shorts. A pattern can be 16X16, 32X32, or 64X64. The origin of the pattern is the lower left corner, so the bottom row is defined first. Each row of the pattern is specified as a short integer. Figure 5.1 shows some possible patterns and their definitions in C. By default, a solid pattern is defined as pattern 0 and this cannot be changed.
defrasterfont (n, ht, nc, chars, nr, rasters)
chars ['g'] = [724, 8, 9, 0, -2, 9]
             raster offset  w  h  xoffset  yoffset  xinc
short raster [ ] = [\ldots]
position 724 > 0x7E00, 0xC300, 0x0300, 0x0300,
                0xF00, 0xC300, 0xC300, 0xC300,
                0x7E00,
                \ldots
}

Figure 5.2
Raster font characters are defined by a bitmap. The width and height of the character, the number of bits in one row of the bitmap, and the baseline position are also specified.
defpattern(n, size, mask)
short n, size;
Ushort mask[];

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

procedure defpattern(n: longint; size: Short; mask: Ibuffer);

The setpattern command selects which of the defined patterns is to be used. The
index provided in the defpattern command is the argument to setpattern. The de-
fault pattern is pattern 0.

setpattern(index)
short index;

subroutine setpat(index)
integer*4 index

procedure setpattern(index: Short);

The getpattern command returns the index of the currently selected pattern.

long getpattern()

integer*4 function getpat()

function getpattern: longint;

5.3 Fonts
defrasterfont defines a raster font and has six arguments. The first argument is
an index into the font table and the second argument is an integer specifying the
maximum height, in pixels, of characters in the font. \texttt{nc} gives the number of
characters in the font and thus the number of elements in the \texttt{chars} array. \texttt{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 left to right and then bottom to top. Mask bits are left-justified in the character's bounding box.

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

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: Short; var chars: Fontchararray;
var raster: Fontraster);
```

One raster font is defined by default. It is a Helvetica-like font with fixed-pitch characters. If the viewport is set to the whole screen, about 110 of these default characters will fit on a line (one character takes nine pixels). If baselines are sixteen pixels apart, 48 lines can be placed on the screen. The default font is font 0 and cannot be redefined.

The font command chooses the font that will be used whenever a charstr command draws a text string. The argument is an index into the font table built with the defrasterfont command. This font remains selected until another font command is executed. By default, font 0 is selected.

```fortran
font(fntnum)
short fntnum;

subroutine font(fntnum)
integer*4 fntnum

procedure font(fntnum: Short);
```

The getfont command returns the index of the currently selected raster font.
long getfont()

integer*4 function getfon()

function getfont: longint;

The getheight command returns the maximum height of a character in the current raster font, including ascenders (present in characters like 't' and 'h') and descenders (as in 'y' and 'p'). The height is returned in pixels.

long getheight()

integer*4 function gethei()

function getheight: longint;

strwidth returns the width (in pixels) of a text string, using the character spacing parameters in the currently selected raster font. This routine is useful only when there is a simple mapping from screen to world space. The user must do the mapping.

long strwidth(str)
String str;

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

function strwidth(str: pstring): longint;
6. Display Modes and Color

The preceding chapters discussed the drawing and attribute commands that determine which pixels are colored to create an image. This chapter discusses how those pixels are colored. When a pixel is drawn, a color value for the pixel is written into the bitplanes. The bitplanes store the screen image created by the drawing commands. There are three display modes which determine how the color values are stored into the bitplanes and how the values are used to display a screen image. A fourth mode, z-buffer mode, is used for hidden surface removal and is discussed in Chapter 12.

6.1 Display Modes

The screen image in an IRIS system is stored in a set of bitplanes (from four to twenty-four). See Figure 6.1. Each bitplane provides one bit of storage per pixel. The corresponding locations in all the bitplanes represent values (from four to twenty-four bits long) for each pixel. These values determine the color of the pixel when it is displayed on the screen. There are three display modes which determine how the values are stored in the bitplanes and how they are used: RGB mode, single buffer mode, and double buffer mode.

The display mode commands discussed in this chapter take effect only after the gconfig command is called. gconfig uses the display mode, the map mode (discussed in the next section), and the number of available bitplanes to define the mapping from colors to bitplanes.

```
gconfig()

subroutine gconfi

procedure gconfig;
```

The color of a pixel is determined by an RGB value. An RGB value consists of three eight-bit intensity values – one for red, one for green, and one for blue. These values are loaded into the three digital-to-analog converters which control the color and brightness of each pixel.

In the first display mode, RGB mode, the value written into the bitplanes when a pixel is drawn is an RGB value. RGB mode is only useful when the system has at least twenty-four bitplanes (room for three eight-bit values). RGB mode is
RGB mode

24-bit RGB value from update controller

24-bit value to display controller

Single buffer mode

12-bit color map index from update controller

Double buffer mode

12-bit color map index from update controller

Figure 6.1

In RGB mode, 24-bit RGB values are stored in the bitplanes. In single and double buffer modes, 12-bit color map indices are stored in the bitplanes.
selected by issuing the `RGBmode` command.

```
RGBmode()

subroutine RGBmod

procedure RGBmode;
```

A more flexible way to use the bitplanes is provided by `single buffer` or `double buffer mode`. In these modes, the values contained in the bitplanes are indexes into a color map. The color map is a table of 24-bit RGB values. This means that a system with fewer than twenty-four bitplanes can still display the full range of colors specified by 24-bit RGB values. For example, although each pixel in a system with four bitplanes can store only sixteen different values, the colors specified in the color map can be any of the $2^{24} = 16,8$ million possible RGB colors. Even more flexibility can be obtained by dividing the color map into sixteen smaller maps. (See the following section for further discussion of color maps.)

In single buffer mode, up to twelve bitplanes (the color map is twelve-bit addressable) are used to store the color map indices. Any extra bitplanes remain unused or are used for z-buffering (see Chapter 12). Images are simultaneously updated and displayed in single buffer mode, so incomplete or changing pictures may appear on the screen. The `singlebuffer` command invokes this display mode.

```
singlebuffer()

subroutine single

procedure singlebuffer;
```

In double buffer mode, the bitplanes are partitioned into two groups, called `front` and `back buffers`. By default, the front buffer is displayed and the back buffer is updated by drawing commands. Thus, a completed image can be displayed while a new one is being drawn.
doublebuffer()

subroutine double

procedure doublebuffer;

The swapbuffers command swaps the front and back buffers during the next vertical retrace period. After an image is drawn in the back buffer, a swapbuffers command displays it.

swapbuffers()

subroutine swapbu

procedure swapbuffers;

swapinterval establishes a minimum time between buffer swaps. If the user specifies a swap interval of 5, the screen will be refreshed at least five times between execution of successive swapinterval commands. This command provides a way to change frames at a steady rate if a new image can be created 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(i)
short i;

subroutine swapin(i)
integer*4 i

procedure swapinterval(i: Short);

It is sometimes convenient to update both the front and the back buffers, or to update the front instead of the back. backbuffer enables updating in the back buffer. If b = TRUE, the default, updating in the backbuffer is enabled; FALSE means it is not enabled.
backbuffer(b)
Boolean b;

subroutine backbu(b)
logical b

procedure backbuffer(b: Boolean);

frontbuffer enables updating in the front buffer. If b = FALSE, the default, the
front buffer is not enabled; TRUE means it is enabled.

frontbuffer(b)
Boolean b;

subroutine frontb(b)
logical b

procedure frontbuffer(b: Boolean);

getbuffer indicates which buffers are enabled for writing. "1", the default, means
that 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;

The getdisplaymode command returns the current display mode, indicated by an
integer: 0 = RGB mode, 1 = single buffer mode, and 2 = double buffer mode.
long getdisplaymode()

integer*4 function getdis()

function getdisplaymode: longint;

The getplanes command returns the number of available bitplanes. For example, a 24-bitplane system returns 24 available planes in RGB mode and 12 available planes in single and double buffer modes. An 8-bitplane system returns 8 available planes in RGB mode, 8 available planes in single buffer mode, and 4 available planes in double buffer mode.

long getplanes()

integer*4 function getpla()

function getplanes: longint;

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

gsync()

subroutine gsync

procedure gsync;

### 6.2 Color Maps

In RGB mode, eight bits each of red, green, and blue intensities are written into the bitplanes. The programmer can choose from a palette of $2^8 = 16.8$ million different colors. The disadvantage of RGB mode is that it is effective only in a system with at least 24 bitplanes. A more flexible technique is the use of a color map. Instead of RGB values, indices into the color map are stored in the bitplanes. The color map stores $2^{12} = 4096$ RGB values. A system with eight bitplanes, for example, can address only $2^8 = 256$ of the 4096 entries in the color map. However, each of those entries has the full 24-bit precision of RGB mode.
The color map can be used in either one of two modes. onemap, the default, organizes the color map as discussed above — a single map with room for 4096 RGB entries.

```
onemap()

subroutine onemap

procedure onemap;
```

multimap organizes the color map as sixteen independent maps, each with a maximum of 256 RGB entries.

```
multimap()

subroutine multim

procedure multim;
```

getcmmode returns the current color map mode: 0 for multimap mode and 1 for onemap mode.

```
long getcmmode()

integer*4 function getcm()

function getcmmode: longint;
```

setmap chooses which of the small maps (0 through 15) is to be used. It is meaningful only in multimap mode.
setmap(mapnum)
short mapnum;

subroutine setmap(mapnum)
integer*4 mapnum

procedure setmap(mapnum: Short);

The `getmap` command returns the number (from 0 to 15) of the currently selected color map. If called in onemap mode, it returns 0.

long getmap()

integer*4 function getmap()

function getmap: longint;

There are two advantages to multimap mode. First, it increases (by up to a factor of sixteen) the number of colors available in a system with less than twelve bitplanes. Second, it provides an additional tool for altering screen images (e.g., an image can be color-inverted simply by switching to a different color map).

`onemap` and `multimap` take effect when `gconfig` is called (see Section 6.1). Note that color map indices are limited to twelve bits in onemap mode and eight bits in multimap mode.

cyclemap cycles through color maps at a selected rate. It specifies a duration (in vertical retraces), the current map, and what map to change to when the duration has timed out. For example, the following commands 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 that program termination does not stop these commands; you must explicitly set all durations to 0.

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

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

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

mapcolor sets a color map entry to a specified RGB value. Its arguments are a
color map index and eight bits each of red, green, and blue intensities. Pixels
written with color color will be displayed with the specified RGB intensities.
(See the following section for a discussion of the color command.) In multimap
mode only the currently selected color map can be updated with mapcolor. In-
valid indices are ignored.

mapcolor(color, red, green, blue)
Colorindex color;
RGBvalue red, green, blue;

subroutine mapcol(color, red, green, blue)
ingeger*4 color, red, green, blue

procedure mapcolor(color: Colorindex; red, green,
blue: RGBvalue);

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

getmcolor(color, r, g, b)
Colorindex color;
RGBvalue *r, *g, *b;

subroutine getmco(color, r, g, b)
ingeger*4 color, r, g, b

procedure getmcolor(color: Colorindex; var r, g, b: RGBvalue);

blink changes the color map entry at a selected rate. It specifies a blink rate, a
color map index, and red, green, and blue values. Every rate vertical retraces,
the color located at index color in the current color map is updated. Its value is
either the original value or the new value supplied by the red, green, and blue arguments. Up to twenty colors can blink simultaneously, each at a different rate. The blink rate can be changed by calling blink again with the same color and a different rate. Calling blink with rate=0 when color specifies a blinking color map entry terminates the blinking and restores the original color.

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

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

procedure blink(rate: Short; color: Colorindex; red, green, blue: RGBvalue);
```

6.3 Colors and Writemasks

In the chapters on drawing primitives and user-defined shapes, the current color is often mentioned. This section describes the command for setting it.

When a pixel is drawn in single or double buffer mode, the current color map index is written into the bitplanes. The color command sets the current index. In onemap mode, the index can be in the range 0-4095. In multimap mode, the index is used in conjunction with a color map number and should be in the range 0-225. The color is also bounded by the number of available planes.

```
color(c)
Colorindex c;

subroutine color(c)
integer*4 c

procedure color(c: Colorindex);
```

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 g1.h and correspond to the color map entries that ginit set to be the colors red and blue. They could, in fact, be set to any of 16.8 million colors.
#include "gl.h"

main()
{
    glinit();
    color(BLUE);
    rectf(0, 0, 100, 100);
    color(RED);
    circl(50, 50, 50);
    glexit();
}

The getcolor command returns the current color. The system must be in single or double buffer mode when this command is executed.

long getcolor()

integer*4 function getcol()

function getcolor: Colorindex;

In RGB mode, the current color is set with the RGBcolor command. The lower-order eight bits of the three arguments are the intensity values for the colors red, green, and blue. These numbers are written into the bitplanes whenever a pixel is drawn; they directly control the intensity of red, green, and blue displayed on the screen.

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

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

procedure RGBcolor(red, green, blue: RGBvalue);

For example, the following program draws a blue rectangle around a red circle 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();
}

The gRGBcolor command returns the current RGB value. The arguments are addresses of three locations that will be filled with the red, green, and blue values. The system must be in RGB mode when this command is executed; otherwise, it is ignored.

```c
short *red, *green, *blue;

subroutine gRGBco(red, green, blue)
integer*2 red, green, blue

procedure gRGBcolor(var red, green, blue: RGBvalue);
```

As discussed so far, a color map index or an RGB value is written into the bitplanes when a pixel is drawn. However, an important tool available to the user is the ability to shield bitplanes from new values. This shielding mechanism allows a layering of images to be created. For example, color values for the cursor can be written into two of the bitplanes in the system. These two bitplanes can then be shielded from ordinary drawing commands (which will write values in the remaining bitplanes). As a result, the cursor will always appear on the screen. See Figures 6.2 and 6.3.

In single buffer or double buffer mode, the writemask command determines which bitplanes will be drawn into by the drawing commands. Its argument is a mask with one bit per available bitplane. The ones in the writemask enable a bitplane for writing. The corresponding bit in the current color index will be written into the bitplane wherever a pixel is drawn. Zeros in the writemask mark bitplanes as read-only. These planes are not changed, regardless of the bits in the color index.
Figure 6.2
Writemasks determine whether or not a new value can be stored in each bitplane. A “1” in the writemask allows a new value (0 or 1) to be stored in the corresponding bitplane. A “0” prevents the new value from being stored and the corresponding bitplane retains its current value. Each a and each b can be either 0 or 1.
Figure 6.3

Writemasks can be used to create a layering of images. 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 color value for that location is determined by the remaining bitplanes.
writemask(wtm)
Colorindex wtm;

subroutine writem(wtm)
integer*4 wtm

procedure writemask(wtm: Colorindex);

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

long getwritemask()

integer*4 function getwri()

function getwritemask: Colorindex;

Similar commands mask bitplanes in RGB mode. The RGBwritemask command takes three arguments, masks for each of three sets of eight planes.

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

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

procedure RGBwritemask(red, green, blue: RGBvalue);

The gRGBmask command returns the current RGB writemask as three eight-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: RGBvalue);

6.4 Cursors

A cursor glyph is a 16x16 array of bits, and is used to show the current screen position of a graphics input device (see Chapter 7). The defcursor command defines an entry in a table of cursor glyphs. As with linestyles and textures, the arguments are a table index and the 16x16 bitmap. By default, one glyph (an arrow) is defined as cursor 0. Figure 6.4 shows some examples of cursor glyphs 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: Short; curs: Cursor);

The setcursor command establishes the characteristics of the cursor. The first argument, index, picks a glyph from the definition table. color and mask set a color map index and writemask for the cursor.
Cursor arrow = \{ 0 \times FE00, 0 \times FC00, 0 \times F800, 0 \times F800, \\
0 \times FC00, 0 \times DE00, 0 \times 8F00, 0 \times 0780, \\
0 \times 03C0, 0 \times 01E0, 0 \times 00F0, 0 \times 0078, \\
0 \times 003C, 0 \times 001E, 0 \times 000E, 0 \times 0004 \}\}

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

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

Figure 6.4
A cursor is a 16×16 array of bits with the origin in the lower left corner.
setcursor(index, color, wtm)
short index;
Colorindex color, wtm;

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

procedure setcursor(index: Short; color, wtm: Colorindex);

The curorigin sets the origin of a cursor (i.e., the position on the cursor that will be aligned with the valuators controlling the cursor position). n is the index of the cursor definition.

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

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

procedure curorigin(n, xorigin, yorigin: Short);

The getcursor command returns the index of the glyph, the color, and the write-mask associated with the cursor, and a Boolean indicating whether the cursor is automatically displayed and updated by the system. The arguments to the command are addresses of four locations where the four cursor attributes are to be returned. The default is the glyph at index 0 in the cursor table, displayed with the color 1, drawn in the first available bitplane, and automatically updated and displayed on each vertical retrace. This command is ignored 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);

In RGB mode, the RGBcursor command allows selection of a cursor glyph from a table of 16x16 bit patterns already defined by the user. The 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: Short; red, green, blue, redm, greenm, bluem: RGBvalue);

The gRGBcursor command returns the six parameters of the RGBcursor command and a Boolean value indicating whether the cursor position is being automatically updated. The system must be in RGB mode when this command is executed.
gRGBcursor(index, red, green, blue, redm, greenm, bluem, b)
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: Short; var red, green, blue, redm,
greenm, bluem: RGBvalue; var b: Boolean);

The cursor, by default, is always displayed. As it is drawn on the screen, the
image that it covers is saved away. When the cursor moves, the saved image is
restored. If the image changes while the cursor is displayed, the saved image
might no longer be valid. This is a concern in single buffer and RGB modes, and
in double buffer mode when the front buffer is enabled. Accordingly, there are
commands to turn the cursor on and off.

cursoff prevents the cursor from being displayed. The command should precede
any drawing commands that might write into the cursor planes.

cursoff()

subroutine cursof

procedure cursoff;

curson requests that the system automatically update and display a cursor. curson
commands are usually paired with cursoff commands, though there is no harm
in calling curson when the automatic cursor is already visible.

curson()

subroutine curson

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

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

The cursor position, whether or not it is visible, is automatically updated to reflect the values of the valuator devices to which it is attached (see Section 7.1 for a discussion of the `attachcursor` command).
7. Input/Output Commands

The IRIS Graphics Library supports three classes of input devices:

- A **valuator** returns an integer value. A dial on a dial and button box is an example of a valuator. A mouse is a pair of valuators: one reports horizontal position, the other vertical position.

- A **button** returns a Boolean value with FALSE = not pushed. Keys on an unencoded keyboard, buttons on a mouse, and switches on a dial and button box are examples of buttons.

- A **keyboard** returns ASCII characters.

Within each class, individual devices are given unique device numbers. Appendix A contains a header file, "device.h", which defines symbolic names for many of the device numbers.

Input devices can be **polled or queued**. Polled devices are interrogated by the user and return the state of the device at the moment of polling. Queued devices make entries in an **event queue**, which is read at the user's convenience. (Queued devices can also be polled at any time.)

Polled devices tend to synchronize the user process and terminal process. The user program detects device events that either happen the instant before the device is polled, or last long enough to be valid the next time the device is polled. For example, a quick click of a button (press and release immediately) is lost if the user program does not poll the button between the press and release.

Queued devices act as asynchronous devices, independent of the user process. Whenever a queued device changes state, an entry is made in the **event queue**. The user program can process the queue in a timely fashion without loss of information. There is, however, some latency involved in reading the queue. All events are entered in the same queue; the search for a particular event may involve reading and servicing many events before the desired one is found.

The user can decide which devices, if any, are queued, and can establish some rules about what constitutes a state change, or **event**, for that device. By default, no devices are queued.

In addition to the input commands, there are commands for controlling the characteristics of the peripheral input/output devices of the IRIS. These commands turn on and off the keyboard click and the keyboard lights, ring the keyboard bell, and control the lights and text on the dial and button box.
<table>
<thead>
<tr>
<th>Device</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOUSE1</td>
<td>right mouse button</td>
</tr>
<tr>
<td>MOUSE2</td>
<td>middle mouse button</td>
</tr>
<tr>
<td>MOUSE3</td>
<td>left mouse button</td>
</tr>
<tr>
<td>MOUSERIGHT</td>
<td>right mouse button</td>
</tr>
<tr>
<td>MOUSEMIDDLE</td>
<td>middle mouse button</td>
</tr>
<tr>
<td>MOUSELEFT</td>
<td>left mouse button</td>
</tr>
<tr>
<td>SW0..SW31</td>
<td>32 buttons on dial and button box</td>
</tr>
<tr>
<td>AKEY..PADENTER</td>
<td>all the keys on the keyboard</td>
</tr>
<tr>
<td>LPENBUT</td>
<td>light pen button</td>
</tr>
<tr>
<td>BPAD0</td>
<td>pen stylus or button for digitizer tablet</td>
</tr>
<tr>
<td>BPAD1</td>
<td>button for digitizer tablet</td>
</tr>
<tr>
<td>BPAD2</td>
<td>button for digitizer tablet</td>
</tr>
<tr>
<td>BPAD3</td>
<td>button for digitizer tablet</td>
</tr>
</tbody>
</table>

Table 7.1: Input Buttons

<table>
<thead>
<tr>
<th>Device</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOUSEX</td>
<td>x valuator on mouse</td>
</tr>
<tr>
<td>MOUSEY</td>
<td>y valuator on mouse</td>
</tr>
<tr>
<td>DIAL0..DIAL8</td>
<td>dials on dial and button box</td>
</tr>
<tr>
<td>LPENX</td>
<td>x valuator on light pen</td>
</tr>
<tr>
<td>LPENY</td>
<td>y valuator on light pen</td>
</tr>
<tr>
<td>BPADX</td>
<td>x valuator on digitizer tablet</td>
</tr>
<tr>
<td>BPADY</td>
<td>y valuator on digitizer tablet</td>
</tr>
<tr>
<td>CURSORX</td>
<td>x valuator attached to cursor (usually MOUSEX)</td>
</tr>
<tr>
<td>CURSORY</td>
<td>y valuator attached to cursor (usually MOUSEY)</td>
</tr>
</tbody>
</table>

Table 7.2: Input Valuators

7.1 Initializing a Device

Valuators are single-valued input devices. The value is a 16-bit integer. Examples are the horizontal and vertical motion of a mouse, and a dial on a dial and button box. The `setvaluator` command assigns an initial value `init` to a valuator. `min` and `max` are lower and upper bounds for the values the device can assume.
setvaluator(v, init, min, max)
Device v;
short init, min, max;

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

procedure setvaluator(v: Device; init, min, max: Short);

The cursor, described in Section 6.4, reflects the position of an input device controlled by the user. Since the cursor moves in two dimensions, its position can be determined by any two valuators. 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 may be used. If desired, the cursor can be controlled by two dials on the optional dial and button box.

The attachcursor command attaches the cursor glyph to the movement of two valuators. It takes two valuator device numbers as its arguments. The first valuator controls the horizontal motion of the cursor; the second argument determines vertical motion.

attachcursor(vx, vy)
Device vx, vy;

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

procedure attachcursor(vx, vy: Device);

7.2 Polling a Device

Once a device has been initialized, it can be polled by the user to determine its current state. Valuators are interrogated with the getvaluator command. Any valuator can be polled, regardless of whether it is queued or not. The argument to getvaluator is a valuator device number. The returned value reflects the current state of the device.
The `getbutton` command polls a button and returns its current state. A device number is supplied as the argument. `getbutton` returns either `TRUE` or `FALSE`, with `TRUE` = pressed.

The `Event Queue` is a mechanism for managing the input devices in a computer system. Each entry in the event queue represents a type and some associated data. The `qdevice` command designates a device as a queued device. Its argument is a device number. An entry is made in the event queue each time the device changes state.

Devices are unqueued using the `unqdevice` command.
unqdevice(v)
Device v;

subroutine unqdev(v)
integer*4 v

procedure unqdevice(v: Device);

Some valuators are "noisy"; an unmoving device may report small fluctuations in value. The noise command allows the user to set a lower limit on what constitutes a move. The value of a noisy valuator v must change by at least \texttt{delta} before the motion is significant. noise determines how often queued valuators will make entries in the event queue. For example, \texttt{noise(v,5)} means that valuator v must move at least 5 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: Short);

A queued button and one or two valuators can be tied together such that whenever the button changes state, both the button change and the current valuator position are recorded in the event queue. The \texttt{tie} command takes three arguments: a button b and two valuators \texttt{v1} and \texttt{v2}. Whenever the button changes state, three entries are made in the queue, recording the current state of the button and the current positions of each valuator. One valuator can be tied to a button by making \texttt{v2 = 0}. A button can be untied from valuators by making both \texttt{v1} and \texttt{v2} 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);

The user can place entries directly into the event queue using the `qenter` command. `qenter` takes two 16-bit integers, `qtype` and `value`, 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: Short);

Three commands read the event queue: `qtest`, `qread`, and `blkqread`. `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: long;

Like `qtest`, `qread` returns the device number of the first entry in the event queue. However, if the queue is empty, it waits until there is an entry in 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;

The blkqread command returns multiple queue entries. The first argument data is an array of shorts, and the second argument n is the length of the array. Each entry consists of two shorts (one is the device number and the other is an associated value), thus the length of the array = \(2^*(\# \text{ of entries})\). blkqread can provide more efficient network communication between an IRIS terminal and a host computer. It can also be used 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: Queuearray, n: Short): longint;

The qreset command removes all entries from the queue and discards them.

qreset()

subroutine qreset

procedure qreset;
7.4 Controlling Peripheral Input/Output Devices

In addition to commands for polling and queuing input devices, there are commands for controlling the characteristics and behavior of the IRIS's peripheral input/output devices.

The `clkon` and `clkoff` commands turn on and off the keyboard click.

```plaintext
clkon()

subroutine clkon
procedure clkon;
```

```plaintext
clkoff()

subroutine clkoff
procedure clkoff;
```

The `lampon` and `lampoff` commands control the four lamps on the keyboard. Each "1" in the four lower-order bits of the `lamps` argument to `lampon` causes the corresponding keyboard lamp to be on. Each "1" in the four lower-order bits of the `lamps` argument to `lampoff` causes the corresponding keyboard lamp to be off.

```plaintext
lampon(lamps)
char lamps;

subroutine lampon(lamps)
integer*4 lamps

procedure lampon(lamps: Byte);
```
lampoff(lamps)
char lamps;

subroutine lampoff(lamps)
integer*4 lamps

procedure lampoff(lamps: Byte);

The **ringbell** command rings the keyboard bell.

ringbell()

subroutine ringbe

procedure ringbell;

The **setbell** command sets the duration of the keyboard bell:

- 0 is off;
- 1 is a short beep;
- 2 is a long beep.

setbell(mode)
char mode;

subroutine setbel(mode)
integer*4 mode

procedure setbell(mode: Byte);

The **dbtext** command writes text to the LED display in a dial and button box. The string *str* must be eight or fewer characters.
dbtext(str)
char str[8];

subroutine dbtext(str)
character*(8) str

procedure dbtext(str: pstring);

The *setblights* command controls the thirty-two lighted switches on a dial and switch box. To turn on switches 3 and 7, for example, *mask* would be \( (1 << 3) | (1 << 7) \).

setblights(mask)
long mask;

subroutine setdbl(mask)
integer*4 mask

procedure setblights(mask: long);
8. Graphical Objects

In a conventional programming language, subroutines can be viewed as semantic extensions. Similarly, graphical objects extend the IRIS Graphics Library. An object is a sequence of graphics commands that is used more than once. By packaging these commands as an object, much of the overhead associated with translating the commands can be avoided. An object is "compiled" into a display list as it is defined. Calls to an object are interpreted directly without further compilation.

8.1 Defining An Object

Objects are created and named with the makeobj command. This command takes a single argument, a 31-bit integer that is the assigned name.

```plaintext
makeobj(obj)
Object obj;

subroutine makeobj(obj)
    integer*4 obj

procedure makeobj(obj: Object);
```

When a makeobj command is executed, the object name is entered into a symbol table and memory is allocated for a display list. Subsequent graphics commands are compiled into the display list instead of being executed.

closeobj terminates the object definition and closes the open object. All the display list commands between makeobj and closeobj become part of the object definition.

```plaintext
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.
The sphere above is defined as a graphical object. The `makeobj` command creates a new object containing all commands between `makeobj` and `closeobj`. Here is the code snippet:

```
makeobj(sphere);
for (phi = 0; phi <= Pi; phi += Pi/18)
  for (theta = 0; theta <= 2*Pi; theta += Pi/18)
    x = sin(theta) * cos(phi);
    y = sin(theta) * sin(phi);
    z = cos(theta);
    move(x,y,z);
else draw(x,y,z);
closeobj();
```
Figure 8.2
Stroked font characters can be graphical objects.

makeobj (g);
circi (15,15,15);
arci (15,3,15,2050,3600);
move2i (38,30);
draw2i (30,30);
draw2i (30,3);
translate (40.,0.,0.);
closeobj ( );
An object defined with a name currently in use replaces the prior contents with the new definition. isobj and genobj ensure that an object name is unique. isobj tests whether an object exists for a given name. If so, the isobj command returns TRUE. It returns FALSE if no object exists with that identification.

```fortran
Boolean isobj(obj)
Object obj;

logical function isobj(obj)
integer*4 obj

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

genobj generates unique object names. It will not generate an object name that is currently in use. genobj is useful in naming objects in library routines, where it is impossible to anticipate what object names will be in use when the routine is called.

```fortran
Object genobj()

integer*4 function genobj()

function genobj: Object;
```

The delobj command deletes an object. All display list memory storage associated with the object is freed, and the object name is marked undefined until it is reused to create a new object. Calls to deleted or undefined objects are ignored.

```fortran
delobj(obj)
Object obj;

subroutine delobj(obj)
integer*4 obj

procedure delobj(obj: Object);
```
8.2 Using Objects

An object is drawn by calling it. The callobj command takes the name of the object to be drawn as its argument.

```plaintext
callobj(obj)
Object obj;

subroutine callobj(obj)
integer*4 obj

procedure callobj(obj: Object);
```
callobj can call one object from inside another object. More complex pictures can be drawn using a hierarchy of simple objects. An example is shown below. The single command callobj (pearls) will draw the object, a string of pearls, by calling the previously defined object pearl seven times.

```plaintext
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 8.3 shows another example using simple objects to build more complex ones. A solar system is defined as an hierarchical object. Calling the one object solarsystem causes all the other objects named in the definition of solarsystem (the sun, the planets, and their orbits) to be drawn.

Global attributes are not saved before a callobj command takes effect. Thus, if attributes like color are changed within an object, the change may affect the caller as well. Use pushattributes and popattributes to preserve global attributes across callobj commands.
Figure 8.3
Solarsystem, a complex object, is defined hierarchically, as shown in the tree diagram. Branches in the tree represent callobj commands.
Figure 8.4
Bounding boxes can be computed to determine which objects are outside the screen viewport. If the bounding box is entirely outside the viewport the object will not be called. The sphere in the bounding box that lies partially within the viewport will be drawn.
Figure 8.5
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.
When a complex object is called, the whole hierarchy of objects in its definition is drawn. The object `solarsystem` in Figure 8.3 is an example. One negative aspect of this is that time may be spent on objects that are not visible in the screen viewport. The `bbox2` command determines whether or not an object is within the viewport or is large enough to be seen. `bbox2` performs the graphical functions known as pruning and culling.

The `bbox2` culling function determines which parts of a picture are less than the minimum feature size, and thus too small to draw on the screen. Its pruning function calculates whether an object is completely outside the viewport.

The `bbox2` command takes as its arguments an object space bounding box, and minimum horizontal and vertical feature sizes, in pixels. The bounding box is calculated, transformed to screen coordinates, and compared with the viewport. If the bounding box is completely outside the viewport, the commands between `bbox2` and the end of the object are ignored. Otherwise, it is compared with the minimum feature size. If the bounding box is too small in both the x and y dimensions, the rest of the commands in the object are ignored. Otherwise, interpretation of the object continues.

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

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

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

Figure 8.4 shows the `solarsystem` discussed previously. The bounding boxes shown in the figure might be used to perform pruning — determining what objects will be even partially in the viewport.

### 8.3 Object Editing

An object can be edited to change its contents. Object editing is useful for complex objects that are time-consuming to recompile. Commands can be added, removed, or replaced through object editing.

`editobj` opens an object for editing. A command pointer acts as a cursor for appending new commands. The command pointer is initially set to the end of the object.
**editobj**(obj)
Object obj;

**subroutine editobj**(obj)
integer*4 obj

**procedure editobj**(obj: Object);

The editing commands following a editobj are interpreted, and the editing session is terminated with a closeobj. If editobj specifies an undefined object, an error message is printed.

Tags within an object make it easy to locate a given command for editing. Editing commands must be given tag names as arguments. It is useful to have tags throughout the object to move to when editing.

Each object has two predefined tags – STARTTAG and ENDTAG. STARTTAG is positioned just after makeobj, before any commands. ENDTAG is always positioned after all the commands in an object, before closeobj. These tags cannot be deleted, and no items can be added before STARTTAG or after ENDTAG. When an object is opened for editing, a pointer is just before ENDTAG, and just after the last command in the object. To perform edits on other lines, refer to them by their tags.

Commands can be explicitly flagged with the maketag command. The user supplies a 31-bit name that is assigned to the command immediately following the maketag command. The tag is local to that object. The same tag name can be used safely in different objects.

**maketag**(t)
Tag t;

**subroutine maketa**(t)
integer*4 t

**procedure maketag**(t: Tag);

newtag also adds tags to an object, but uses an existing tag to determine its relative position in the object. newtag creates a new tag offset the number of lines given as its argument beyond the other tag.
newtag(newtag, oldtag, offset)
Tag newtag, oldtag;
long offset;

subroutine newtag(newtag, oldtag, offset)
integer*4 newtag, oldtag, offset

procedure newtag(newtag, oldtag: Tag; offset: longint);

istag tells whether a given tag is in use within the currently open object. istag
returns TRUE if the tag is in use, and FALSE if it is not. The result is undefined
if there is currently no open object.

Boolean istag(t)
Tag t;

logical function istag(t)
integer*4 t

function istag(t: Tag): longint;

gentag generates a unique integer for use as a tag.

Tag gentag()

integer*4 function gentag()

function gentag: Tag;

deltag deletes tags from the object currently open for editing. The special tags
STARTTAG and ENDTAG cannot be deleted.
deltag(t)
Tag t;

subroutine deltag(t)
   integer*4 t

   procedure deltag(t: Tag);

objinsert adds commands to an object at the given location. objinsert takes a tag as an argument, and positions an editing pointer on that tag. Graphics commands are added immediately after the tag. To terminate the insertion, use closeobj or another editing command (objdelete, objinsert, objreplace).

objinsert(t)
Tag t;

subroutine objins(t)
   integer*4 t

   procedure objinsert(t: Tag);

objdelete removes commands from the currently open object. It removes everything between the two tagnames given as arguments – both commands and other tagnames are deleted. For example: objdelete(STARTTAG, ENDTAG) would delete every line (except makeobj() and closeobj() ) in the object specified. The command is ignored if no object is open for editing. This command leaves the pointer at the end of the object after it takes effect.

objdelete(tag1, tag2)
Tag tag1, tag2;

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

   procedure objdelete(tag1, tag2: Tag);

objreplace combines the functions of objinsert and objdelete. This command is a quick way to replace a command with a version of itself containing different arguments. It takes as its argument a single tag. Graphics commands that follow
objreplace overwrite existing commands until a closeobj or editing command (objinsert, objreplace, objdelete) terminates the replacement. objreplace requires the new command to be exactly the same length in characters as the one replaced. This makes objreplace operations faster, but more general replacement should be done with objdelete and objinsert commands.

```ruby
objreplace(t)
Tag t;

subroutine objrep(t)
integer*4 t

procedure objreplace(t: Tag);
```

Here is an example of object editing. An object star is defined:

```ruby
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 commands to give a modified object:

```ruby
editobj(star);
circi(1, 5, 5);
objinsert(BOX);
recti(0, 0, 10, 10);
objreplace(INNER);
color(GREEN);
closeobj();
```
The resulting object after the editing session:

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

getopenobj determines whether any objects are currently open for editing. If an object is open, it returns its number. If no objects are open, it returns -1.

---

Object memory management

Three commands perform necessary memory management tasks: compactify, chunksize, and getmem.

An open object may become fragmented and be stored inefficiently in memory as it is modified by the various editing commands. When the amount of wasted space becomes large, compactify is automatically called during the closeobj operation.

This command allows the user to perform the compaction explicitly. Unless new commands have been inserted, compaction is not necessary. Since compactify uses a significant amount of computing time, it should not be called unless storage space is critical.
compactify(obj)
Object obj;

subroutine compac(obj)
integer*4 obj

procedure compactify(obj: Object);

If there is a memory shortage, chunksize can allocate memory in standard amounts (chunks) to an object. chunksize specifies the minimum amount of memory allocated to an object, and as the object grows, it is allocated more memory in units of size chunk. chunksize may be called only once after ginit and before the first makeobj. The default chunk unit of size is 1020 bytes.

chunksize(chunk)
long chunk;

subroutine chunks(chunk)
integer*4 chunk

procedure chunksize(chunk: longint);

getmem determines the amount of available memory left on the system. 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;

finish is used only on an IRIS terminal running remote graphics. finish blocks the host process until all prior commands have been executed. It forces all unsent commands 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 an acknowledgement has been sent. Network and pipeline delays make this command useful.
finish()

subroutine finish

procedure finish;

callfunc allows an arbitrary function call from within an object. When it is executed 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.

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

NOTE: works only in C

NOTE: works only in C
9. Picking and Selecting

Previous chapters have discussed how to define objects in world coordinates and then draw them to the screen. This chapter discusses how to specify an area of the screen (or of the "world") and then determine what objects are being drawn in that area.

The mapw command takes 2D screen coordinates and identifies the corresponding 3D world coordinates. The pick command puts the IRIS in picking mode – all the objects drawn to a small area around the cursor can be identified. The select command puts the IRIS in selecting mode – all the objects drawn to a 3D area (specified by the user) can be identified.

9.1 Mapping the Screen to World Space

The mapw command takes a 2D screen point and maps it onto a line in 3D world space. The viewing, projection, and viewport transformations that map the currently displayed objects to the screen are contained in the viewing object vobj. mapw reverses these transformations and maps the screen space point back to a world space line. 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.

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

mapw2 is the two-dimensional version of mapw. In the 2D case, a screen point is mapped to a world space point rather than a line. Again, vobj contains the projection and viewing transformations that map the displayed objects to world space, and sx and sy define a screen space point. The corresponding world space coordinates are returned in wx and wy. If the transformations in vobj are not 2D (i.e., not orthogonal projections), the result is undefined.
Figure 9.1
In picking mode, the user 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 would be picked. The text string whose origin was in the picking region would also be picked. The shaded box and the other text string would not be picked.
map2(obj, sx, sy, wx, wy)
Object obj;
Screencoord sx, sy;
Coord *wx, *wy;

subroutine map2(obj, sx, sy, wx, wy)
integer*4 obj, sx, sy
real wx, wy

procedure map2(obj: Object; sx, sy: Screencoord; var wx, wy: Coord);

9.2 Picking

Picking allows the user to identify objects by pointing at them with an input device. The pick command puts the IRIS in picking mode.

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

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

procedure pick(buffer: Ibuffer; numnames: longint);

When a command is called in picking mode, the screen does not change. Instead, the IRIS determines whether or not the command is drawing into a small rectangular area around the cursor called the picking region. Any command that does draw within the picking region is a "hit". (charstr causes a hit only when the origin of the string is located in the picking region.) Figure 9.1 shows four objects and a cursor glyph; only two of the objects would be picked if they were called during picking mode. Notice that clear() always causes a hit – it always draws in the picking region.

When a hit occurs, the contents of the name stack are copied into a buffer. It is the user's responsibility to maintain an informative list of names in the name stack. The commands that manipulate the name stack and examples of their use are presented below. At the end of a picking session, the picking buffer contains a series of name-lists, one list for each hit. The numnames argument to pick
specifies the maximum number of values that can be stored in the buffer.

The user maintains the name stack with the loadname, pushname, popname, and initnames commands. These commands can be interspersed with drawing commands while in picking mode or they can be inserted into object definitions. See Chapter 8 for a discussion of objects.

loadname puts name at the top of the name stack and erases whatever was there before.

```verbatim
loadname(name)
short name;

subroutine loadna(name)
integer*4 name

procedure loadname(name: Short);
```

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

```verbatim
pushname(name)
short name;

subroutine pushna(name)
integer*4 name

procedure pushname(name: Short);
```

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

```verbatim
popname()

subroutine popnam

procedure popname;
```

initnames discards all the names in the stack and leaves the stack empty.
initnames()

subroutine initna

procedure initnames;

Each name in the name stack is sixteen bits long. The maximum number of
names that can be stored in the name stack is 1000.

The endpick command takes the IRIS out of picking mode and returns the
number of hits that occurred. If the returned number is positive, then there was
space in the buffer for all of the name-lists. If the returned number is negative,
then the buffer was too small to contain all the name-lists and the magnitude of
the returned number is the number of name-lists that were stored.

long endpick(buffer)
short buffer[];

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

function endpick(var buffer: Short): longint;

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

The default height and width of the picking region is ten pixels. The picking re-

gion can be changed with the pickszie command. deltax and deltay specify a rec-
tangle centered at the current cursor position (the origin of the cursor glyph).
(See Section 6.4 for a discussion of cursors.)
picksize(delta_x, delta_y)
short delta_x, delta_y;

subroutine picksi(delta_x, delta_y)
integer*4 delta_x, delta_y

procedure picksize(delta_x, delta_y: Short);

Picking loads a projection matrix that makes the picking region fill the entire viewport. This picking matrix replaces the projection transformation matrix that would normally be used when the drawing commands are called. Therefore, the original projection transformation must be restated after the pick command (to ensure that the objects to be picked will be mapped to the proper coordinates). If no projection transformation was originally issued, the default ortho2 must be specified. When the transformation command is restated, the product of the transformation matrix and the picking matrix is placed at the top of the matrix stack. If the projection transformation is not restated, picking does not work correctly. (Typically, every object will be picked, regardless of cursor position and picksize.)

The following program draws an object consisting of three shapes and then loops until the right mouse button is pressed. Each time the middle mouse button is pressed, the IRIS enters pick mode, calls the object, records hits for any commands that draw into the picking region, and prints out the contents of the picking buffer. Note that when the object is called in pick mode, the screen does not change. Since the picking matrix is recalculated only when pick is called, picking mode is exited and reentered to obtain new cursor positions.

#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" */
circl(50,500,50);
loadname(4); /* replace "3" with "4", so the stack has "4 2" */
circl(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 */
        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 */
        printf("hits: %d, \n", 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");
    }
}
Here is the FORTRAN version of the program:

```
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
C Installation note:
C Various Fortran compilers may require different styles of INCLUDEs.
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
INCLUDE 'fg1.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(50,530,50)
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, IRIS enters pick mode
IF (TYPE .EQ. MOUSE2) THEN
CALL PICK(NAMBUF,50)
C  restate the projection transformation for 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 each hit
NUMPIC = ENDPIC(NAMBUF)
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

There are five possible outcomes for each picking session (the circles can be picked together because they overlap):

1) nothing is picked = "hits: 0;"
2) the rectangle is picked = "hits: 1; 1 1 1"
3) the first circle is picked = "hits: 1; 2 3 2 1"
4) the second circle is picked = "hits: 1; 2 4 2 1"
5) both the first and second circle are picked = "hits: 2; 2 3 2 1 2 4 2 1"
In addition to identifying the command that caused a hit, the IRIS provides a code that indicates which of the picking region's clipping planes were intersected by the last hit in a picking session. The *hitcode* is a six-bit number with one bit for each clipping plane, as shown in the following table.

<table>
<thead>
<tr>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>far</td>
<td>near</td>
<td>top</td>
<td>bottom</td>
<td>right</td>
<td>left</td>
</tr>
</tbody>
</table>

After *endpick* has been called, the *gethitcode* command can be called to obtain the hitcode for picking session. "1" in the hitcode indicates that the corresponding plane was intersected during the picking session. "0" indicates the plane was not intersected. (Systems without near-far clipping always return "0"s for the near and far clipping planes.)

```plaintext
long gethitcode()

integer*4 function gethit()

function gethitcode: longint;
```

The *clearhitcode* command sets all the bits in the current hitcode to "0". It is used before the picking session begins to initialize the hitcode.

```plaintext
clearhitcode()

subroutine clearh

procedure clearhitcode;
```
Figure 9.2
In selection mode, the user can identify all commands that draw things in an arbitrary region of world space. Here, we can detect that the rocket ship will crash into the planet by defining the selection region as a box around the ship.
9.3 Selecting

Selecting is a more general mechanism than picking for identifying the commands that draw to a particular region. A selecting region is a two- or three-dimensional area of world space. When the `select` command turns on selecting mode, the region represented by the current viewing matrix becomes the selecting region. The selecting region can be changed at any time by issuing a new viewing transformation command. To use selecting mode, issue a viewing transformation command that specifies the selecting region, call `select`, call the objects or commands of interest, exit selecting mode, and look to see what was selected.

```
select(buffer, numnames)
short buffer[0];
long numnames;

subroutine select(buffer, numnam)
integer*2 buffer(0)
integer*4 numnam

procedure select(buffer: Ibuffer; numnames: longint);
```

`numnames` specifies the maximum number of values that can be stored in the buffer. The name stack and the hitcode commands are used in the same way as picking. `endselect` turns off selecting mode. A series of name-lists is reported to the user in `buffer`. Each name-list represents the contents of the name stack when a command was called that drew into the selecting region. The number of name-lists in `buffer` is returned as the value of `endselect`. If the number is negative, more commands 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: Ibuffer): integer;
```

Figure 9.2 shows a sample application of 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.

```c
#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);
    cursoroff();
    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();

select(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[i]==1) {
    printf("CRASH\n");
curscn();
gexit();
}

gexit();
}
10. Geometry Pipeline Feedback

The IRIS feedback facility provides access to the Geometry Pipeline for matrix multiplication and geometric computing. In feedback mode, the data output by the Geometry Pipeline are stored in a buffer, rather than being sent to the raster display subsystem. The buffer can then be examined to see how the data were transformed by the Geometry Engines. For example, if the \texttt{pat} command is issued in feedback mode, the Geometry Pipeline performs its normal function of transforming, clipping, and scaling the world coordinates to screen coordinates. The command code for the \texttt{pat} command, along with the new transformed coordinates, is then stored in the feedback buffer. (If the point is outside the clipping region, no command is output from the Geometry Pipeline and nothing is stored in 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 what is saved in the buffer.

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 issued by the processor can be divided into two classes—those that are interpreted by the Geometry Pipeline and those that are passed through unchanged. The commands that are interpreted by the Geometry Pipeline include the move, draw, point, polygon, curve, and matrix manipulation commands. Commands not interpreted by the Geometry Pipeline include (among others) the color, character string, texture, linestyle, and display mode commands.

The Geometry Pipeline consists of two geometry accelerators and ten or twelve Geometry Engines. (See Figure 10.2.) There are three Geometry Pipeline subsystems:

- The \textit{matrix subsystem} consists of four matrix multipliers, one for each column in a 4x4 matrix.

- The \textit{clipping subsystem} consists of four or six clippers, one for each clipping plane of the drawing space.

- The \textit{scaling subsystem} consists of two scalers, which scale 2D and 3D coordinates to the coordinate system of a particular output device.

Each Geometry Engine can be configured to do any of the twelve possible jobs. (Systems with ten Geometry Engines have only four clippers and do not clip to the near and far planes.)
Figure 10.1
In feedback mode, the output of the Geometry Pipeline is returned to the Applications/Graphics Processor.
Figure 10.2
The Geometry Pipeline consists of two Geometry Accelerators and three Geometry Engine subsystems: the matrix multipliers, the clipplers, and the scalers.
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 to make 4-dimensional homogeneous coordinates. The first four Geometry Engines – the matrix multipliers – transform the coordinates to a normalized screen space. Each of these four engines performs the calculations for one column of the current transformation matrix. The current transformation matrix is computed from the modeling, viewing, and projection transformation commands issued by the user, and is automatically loaded into the Geometry Engines. The user can directly define the transformation matrix in the Geometry Engines with the matrix manipulation commands (e.g., loadmatrix) discussed in Chapter 4.

The next four Geometry Engines clip the normalized screen coordinates to the top, bottom, left, and right clipping planes. Two optional Geometry Engines clip to the near and far planes. The clipping process may cause commands to be clipped out entirely or it may cause new commands to be generated. Figure 10.3(a) shows how clipping can cause a new move command to be inserted in the command stream. Figure 10.3(b) shows how a three-point polygon turns into a seven-point polygon after being pass through the clippers.

The final two Geometry Engines – the scalers – map the normalized, clipped screen-space coordinates to a particular area of the screen. This area of the screen is defined by the viewport command. (See Chapter 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 the user to perform matrix operations and geometric computing with the Geometry Pipeline. During feedback mode, the output of the Geometry Pipeline (a series of 16-bit integers) is saved in a buffer. Each command appears as a token, followed by data values. Table 10.1 shows the tokens for commands that are transformed by the Geometry Pipeline, along with the data associated with each command. Note that circles, arcs, polygons, curves, and patches all appear in the feedback buffer as move’s and draw’s. Most attribute and viewport commands appear in the feedback buffer but should be avoided in feedback mode since they are not transformed and complicate the interpretation of the buffer. The passthrough and xft commands at the bottom of the table are discussed later in this section.
<table>
<thead>
<tr>
<th>command</th>
<th>normal mode</th>
<th>depth-cue mode</th>
<th>z-buffer mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>move</td>
<td>16, x, y</td>
<td>16, x, y, x', z</td>
<td></td>
</tr>
<tr>
<td>draw</td>
<td>17, x, y</td>
<td>17, x, y, x', z</td>
<td></td>
</tr>
<tr>
<td>pnt</td>
<td>18, x, y</td>
<td>18, x, y, x', z</td>
<td></td>
</tr>
<tr>
<td>pmov</td>
<td>48, x, y</td>
<td>48, x, y, x', z</td>
<td></td>
</tr>
<tr>
<td>pdr</td>
<td>49, x, y</td>
<td>49, x, y, x', z</td>
<td></td>
</tr>
<tr>
<td>pclos</td>
<td>51, x, y</td>
<td>51, x, y, x', z</td>
<td></td>
</tr>
<tr>
<td>passthrough</td>
<td>8, value</td>
<td>8, value</td>
<td></td>
</tr>
<tr>
<td>xfpt</td>
<td>56, x, y, z, w</td>
<td>56, x, y, z, w</td>
<td></td>
</tr>
</tbody>
</table>

Table 10.1: Command tokens and associated data

The feedback command puts the IRIS in feedback mode. The first argument to feedback is the name of the buffer where the Geometry Pipeline output is saved. The second argument specifies the numbers of items to be saved by the system. If more than size values appear, the extra values will be lost.

```c
feedback(buffer, size)
short buffer[];
long size;

subroutine feedback(buffer, size)
integer*2 buffer(*)
integer*4 size

procedure feedback(buffer: Ibuffer; size: longint)
```

The endfeedback command returns the number of values stored in the buffer.

```c
long endfeedback(buffer)
short buffer[];

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

function endfeedback(buffer: Short): longint;
```

The passthrough command 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 and can be used to match the input to the pipeline with the out-
The sequence: move(\textless A\textgreater)  
move(\textless A\textgreater)  
draw(\textless B\textgreater)  
draw(\textless C\textgreater)  
becomes:  
move(\textless 1\textgreater)  
draw(\textless 2\textgreater)  
move(\textless 3\textgreater)  
draw(\textless 4\textgreater)  

The sequence: pmove(\textless A\textgreater)  
pdraw(\textless B\textgreater)  
pdraw(\textless C\textgreater)  
pclose  
becomes:  
pmove(\textless A\textgreater)  
pdraw(\textless 1\textgreater)  
pdraw(\textless 2\textgreater)  
pdraw(\textless 3\textgreater)  
pdraw(\textless 4\textgreater)  
pdraw(\textless 5\textgreater)  
pdraw(\textless 6\textgreater)  
pclose  

Figure 10.3  
Two examples of how the clipping subsystem of the  
Geometry Pipeline can generate new commands.
put from the pipeline. _passthrough_ is especially useful for identifying commands that are clipped and do not appear in the feedback buffer.

```
passthrough(token)
short token;

subroutine passth(token)
integer*4 token

procedure passthrough(token: Short);
```

The following program issues three _pas_ 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.

```
#include "gl.h"

main()
{

    unsigned short buf[200];
    register i,j,num;
    float *bufptr;

    glinit();
    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" */
    pnt2i(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("%d\n");
        printf("%0.4fx%0.4ft", buf[i]);
    }
    printf("\n");
    printf("\n");
```
gexit();
}

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 2 3 6 8 2
18 2 9 1 5 6 4
```

In order to compute and save z coordinates in the feedback buffer, the IRIS can be put in z-buffer mode or depth-cue mode. (See Chapters 12 and 13.) In these modes, the output of the Geometry Pipeline is five values for each command: a command code, an x coordinate, a y coordinate, another x coordinate (used for stereo projection), and a z coordinate. (See Table 10.1.) The following program issues three pat commands with both feedback mode and depth-cue mode turned on, and prints out the contents of the feedback buffer.

```c
#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]);
    }
}```


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

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
9 1
18 1 2 1 1 0
9 2
18 291 564 291 5
```

The first number in each list is a command code. For the `pat` commands, the following four values are x, y, x', and z.

In the first example, the coordinates for the points were transformed by the matrix multipliers, and then clipped and scaled. However, it is possible to transform coordinates with the matrix multipliers and then bypass the clipplers and scalers. The `xfpt` command maps 4-dimensional points to new 4-dimensional points, using the current transformation matrix. The Geometry Pipeline outputs the `xfpt` command code 56 and four floating-point values, which are saved in the feedback buffer. `xfpt` always stores a four-dimensional transformed floating-point coordinate in the feedback buffer. The input for `xfpt` can be expressed in integer/floating-point/short and 2D/3D/4D versions. `xfpt` should only be called in feedback mode.

---

```c
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 issues three `xfpt` commands:

```c
#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("%0.4f\t", buf[i]);
    }
    printf("\n");
    printf("\n");
    /* print out floating-point versions of the coordinates */
    for (i = 0; i < 3; i++) {
        bufptr = (float *)&buf[i+9]);
        for (j = 0; j < 4; j++)
            printf("%f\t", *bufptr);
        printf("\n");
    }
    exit();
}

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
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 commands for representing 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, and
- a basis which defines how the control points will be used 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 sixteen control points,
- the number of curve segments to be drawn in each direction of the patch, and
- the two bases which 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. In order to create smooth joints, it is necessary to control the positions and curvatures at the endpoints of curve segments. Parametric cubic curves are the lowest-order representation of curve segments that can 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 some 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 $0 \leq t \leq 1$), 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, one constraint might be that one endpoint of the curve segment is located at the first control point. Another constraint could be that 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 point values, 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.\textsuperscript{1}) The set of constraints which define each class is given, along with the basis matrix derived from those constraints. The next section discusses how to use the basis matrices to draw curve segments.

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:

1) One endpoint of the segment is located at \( p_1 \):

   \[
   \text{Bezier}(0) = p_1
   \]

2) The other endpoint is located at \( p_4 \):

   \[
   \text{Bezier}(1) = p_4
   \]

3) The first derivative, or slope, of the segment at one endpoint is equal to this value:

   \[
   \text{Bezier}(0)' = 3(p_2 - p_1)
   \]

4) The first derivative at the other endpoint is equal to this value:

   \[
   \text{Bezier}(1)' = 3(p_4 - p_3)
   \]

Solving for these constraints yields the following equation:

\[
\begin{bmatrix}
-1 & 3 & -3 & 1 \\
3 & -6 & 3 & 0 \\
-3 & 3 & 0 & 0 \\
1 & 0 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
p_1 \\
p_2 \\
p_3 \\
p_4
\end{bmatrix}
= T M_p G_b
\]

All the points on the Bezier cubic curve segment from \( p_1 \) to \( p_4 \) can be generated by evaluating \( \text{Bezier}(t) \) for \( 0 \leq t \leq 1 \). (It is more efficient, however, to construct

Figure 11.1
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.
a forward difference matrix which can generate the points in a curve segment incrementally. Forward difference matrices are discussed in the next section.)

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. The technique of overlapping sets of control points can be used more effectively with the following two classes of cubic curves to create a single large curve from a series of curve segments.

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, referenced above.

The Cardinal spline basis matrix is derived from the following four constraints:

\[
\text{Cardinal} (0) = p_2
\]

\[
\text{Cardinal} (1) = p_3
\]

\[
\text{Cardinal} (0) \cdot a = (p_3-p_1)
\]

\[
\text{Cardinal} (1) \cdot a = (p_4-p_2)
\]

The scalar coefficient \( a \) must be positive, and determines the length of the tangent vector at point \( p_2 \): \( \text{tangent}_2 = a(p_3-p_1) \) and \( \text{tangent}_3 = a(p_4-p_3) \). Solving for these constraints yields the following equation:

\[
\text{Cardinal} (t) = \begin{bmatrix}
-2 & -2 & 1 & 1 \\
-3 & 3 & 0 & 0 \\
-a & a & 0 & 0 \\
0 & 0 & 1 & 0
\end{bmatrix} \begin{bmatrix}
p_0 \\
p_1 \\
p_2 \\
p_3 \\
p_4
\end{bmatrix}
\]

\[
= TM_t G_c
\]

The three joined Cardinal spline curve segments in Figure 11.1 use the same three sets of control points as the Bezier curve segments.

A **B-spline** curve segment does not in general 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:

\[
B\text{-spline}(0)^{\cdot} = \frac{(p_3-p_1)}{2}
\]

\[
B\text{-spline}(1)^{\cdot} = \frac{(p_4-p_2)}{2}
\]

\[
B\text{-spline}(0)^{\cdot\cdot} = p_1-2p_2+p_3
\]

\[
B\text{-spline}(1)^{\cdot\cdot} = p_2-2p_3+p_4
\]

Solving for these constraints yields the following equation:

\[
B\text{-spline}(t) = \left[t^3 t^2 t^1 1\right] \frac{1}{6} \begin{bmatrix}
-1 & 3 & -3 & 1 \\
3 & -6 & 3 & 0 \\
-3 & 0 & 3 & 0 \\
1 & 4 & 1 & 0
\end{bmatrix} \begin{bmatrix}
p_1 \\
p_2 \\
p_3 \\
p_4
\end{bmatrix}
\]

\[= T M_n G_n \]
11.2 Drawing Curves

Drawing a curve segment on the screen involves four steps:

1) The `defbasis` command defines and names a basis matrix.

```plaintext
defbasis(id, mat)
    short id;
    Matrix mat;

    subroutine defbas(id, mat)
        integer*4 id
        real mat(4,4)
    procedure defbasis(id: integer; mat: Matrix);
```

2) The `curvebasis` command selects a defined basis matrix as the current basis matrix.

```plaintext
curvebasis(basisid)
    short basisid;

    subroutine curveb(basisid)
        integer*4 basisid
    procedure curvebasis(basisid: integer);
```

3) The `curveprecision` command specifies the number of line segments to be used to approximate each curve segment.
curveprecision(nsegments)
short nsegments;

subroutine curvep(nsegments)
integer*4 nsegments

procedure curveprecision(nsegments: integer);

4) The *crv* command 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(geom: Curvarray);

When the *crv* command is issued, a matrix is built from the geometry, the current basis, and the current precision:

\[ M = F_{\text{precision}} M_{\text{basis}} G_{\text{geom}} \]

\[
\begin{bmatrix}
\frac{6}{n^3} & 0 & 0 & 0 \\
\frac{6}{n^2} & 2 & 0 & 0 \\
\frac{1}{n^2} & \frac{1}{n} & \frac{1}{n} & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

where \( n \) = the current precision. The bottom row of the resulting transformation matrix identifies the first of \( n \) points that will 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,
Figure 11.2
Each of the above curve segments uses the same set of four control points and the same precision, but a different basis matrix.
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.

```c
/* This is the forward difference algorithm */
/* W is the current transformation matrix */
move (W[3][0]/W[3][3], W[3][1]/W[3][3], W[3][2]/W[3][3]);
/* iteration loop */
for (cnt = 0; cnt < iterationcount; cnt++) {
    for (i=3; i>0; i--)
        for (j=0; j<4; j++)
            W[i][j] = W[i][j] + W[i-1][j];
    draw(W[3][0]/W[3][3], W[3][1]/W[3][3], W[3][2]/W[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: \((0, 0, 0, 1); \left(\frac{1}{n}, \frac{1}{n}, \frac{1}{n}, 1\right); \left(\frac{2}{n}, \frac{1}{n}, \frac{1}{n}, 1\right); \left(\frac{1}{n}, \frac{3}{n}, \frac{3}{n}, 1\right); \ldots\). This is the same sequence of points as is 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 `beplinemat`) contain the values discussed in the previous section.

```c
#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, 0.5, 0.5, 0},
    {0, 1, 0, 0}
};

Matrix beplinemat = {
    {-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}
};
```
```c
#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();
}
```

Here is the FORTRAN version of the program:
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,
  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 GEOM1 /100.0, 100.0, 0.0,
  200.0, 200.0, 0.0,
  200.0, 0.0, 0.0,
  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
        CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
        C    Installation note:
        C    This is a delay loop, which may require adjustment.
        CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
        CALL GFLUSH()
        DO 1000 I=1,1000000
          NOP = I * 2
    1000    CONTINUE
        CALL GEXIT()
        STOP
        END

The **crvn** command takes a series of n control points and draws a curve segment using the current basis and precision. Calling **crvn** has the same effect as calling a sequence of **crv** commands 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; geom: Coord3array);
```

Note that a **crvn** command issued with a Cardinal spline or B-spline basis produces a single curve. A **crvn** command issued with a Bezier basis, however, pro-
Figure 11.3
Each patch is drawn using the same set of sixteen control points, the same number of curve segments, the same precisions, but different basis matrices.
Figure 11.4
The above surface consists of three joined patches. The patches are drawn with overlapping sets of control points, using a Cardinal spline basis matrix.
duces several separate curve segments.

The following program draws the three joined curve segments in Figure 11.1 using the crvn command. geom2 contains six control points.

```
#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 */
```
curvabasis(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 */

curvabasis(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 */

curvabasis(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();

The iteration loop of the forward difference algorithm is implemented in the
Geometry Pipeline. The curveit command 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) iteration-
count times. Each iteration draws one of the line segments that approximate the
curve. The curveit command does not execute the initial move command in the
forward difference algorithm. A move(0.0,0.0,0.0) command must precede the
curveit command so that the correct first point will be generated from the for-
ward difference matrix.

curveit(iterationcount)
short iterationcount;

subroutine curveit(count)
integer*4 count

procedure curveit(iterationcount: Short);

The following program draws the Bezier curve segment in Figure 11.2 using the
curveit command. The Cardinal spline and B-spline curve segments could be
drawn using a similar sequence of commands — only the basis matrix would be
different.
#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()
{
    glinit();
    color(BLACK);
    clear();
    pushmatrix(); /* the current transformation matrix on the
    matrix stack is saved */
    multmatrix(geom1); /* the product of the current transformation
    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 */
    exit();
}
11.3 Drawing Surfaces

The method for drawing surfaces is similar to that of drawing curves. A surface patch appears on the screen as a "wireframe" of curve segments. The shape of the patch is determined by a set of user-defined control points. 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:

\[
x(s,t) = a_{11}s^3t^3 + a_{12}s^3t^2 + a_{13}s^3t + a_{14}s^3 + a_{21}s^2t^3 + a_{22}s^2t^2 + a_{23}s^2t + a_{24}s^2 + a_{31}st^3 + a_{32}st^2 + a_{33}st + a_{34}s + a_{41}t^3 + a_{42}t^2 + a_{43}t + a_{44}
\]

(The equations for \( y \) and \( z \) are similar.) The points on a bicubic patch are defined by varying the parameters \( s \) and \( t \) 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 \( s \) constant at several values and using the IRIS curve facility to draw curve segments in one direction, and then doing the same for \( t \) in the other direction.

There are five steps involved in drawing a surface patch:

1) The appropriate curve bases are defined using the `defbasis` command. (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 the `patchbasis` command. Note that the \( u \) basis and the \( v \) basis do not have to be the same.
patchbasis(uid, vid)
long uid, vid;

subroutine patchb(uid, vid)
integer*4 uid, vid

procedure patchbasis(uid, vid: longint);

3) The number of curve segments to be drawn in each direction is specified by the `patchcurves` command. A different number of curve segments can be drawn in each direction.

patchcurves(ucurves, vcurves)
long ucurves, vcurves;

subroutine patchc(ucurves, vcurves)
integer*4 ucurves, vcurves

procedure patchcurves(ucurves, vcurves: long);

4) The precisions for the curve segments in each direction must be specified with the `patchprecision` command. 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 that the $u$ and $v$ curve segments which form the wireframe actually intersect.

patchprecision(usegments, vsegments)
long usegments, vsegments;

subroutine patchp(usegments, vsegments)
integer*4 usegments, vsegments

procedure patchprecision(usegments, vsegments: long);

5) The surface patch is actually drawn with the `patch` command. The arguments to `patch` contain the sixteen control points that govern the shape of the patch. `geomx` is a 4×4 matrix containing the x coordinates of the sixteen
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.

```
patch(geomx, geomy, geomz)
Matrix geomx, geomy, geomz;

subroutine patch(geomx, geomy, geomz)
real geomx(4,4), geomy(4,4), geomz(4,4)

procedure patch(geomx, geomy, geomz: Patcharray);
```

The following program draws three surface patches similar to those shown in Figure 11.3. All three use the same set of sixteen control points.

```
#include "gl.h"

Matrix beziernmatrix = {
    { -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 }
};

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 beplinematrix = {
    { -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
ucurves greater than usegments) and the curve
segments in the v direction will consist of 21
line segments (the lowest multiple of ucurves
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();
}

Here is the FORTRAN version of the program:

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

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 u curves greater than usegments)
C & the curve segments in the v direction will consist of 21 line
C segments (the lowest multiple of u curves 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   reset the bases for both directions again
CALL PATCHB(BSPLIN, BSPLIN)
CALL COLOR(BLUE)

C   draw a third patch
CALL PATCH(GEOMX, GEOMY, GEOMZ)

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C   Installation note:
C   This is a delay loop, which may require adjustment.
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CALL GFLUSH()
DO 1000 I=1,1000000
NOP = I * 2
1000 CONTINUE
CALL GEXIT()
STOP
END

Patches can be joined 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.
12. Hidden Surfaces

Chapter 4 (Coordinate Transformations) discussed how to create 3D models in world space and how to project images of those models onto the screen. This chapter discusses how to make those images more realistic by removing the hidden lines and surfaces from the images: 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 for a discussion of bitplanes and display modes.) When a pixel is to be 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. The result is that only the parts of the image visible to the viewer are displayed on the screen.

Z-buffering only works in single buffer display mode. The z-buffer bits occupy the sixteen high-order bitplanes of a 32-bitplane system and the twelve 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.

Three commands set up z-buffering:

1) The `setdepth` command sets the scaling of the near and far clipping planes. `setdepth` specifies the range of z values that will be stored in bitplanes. If there are sixteen bits of z-buffer, `near` should be set to 0x8000 (the smallest value that can be stored in the sixteen high-order bitplanes) and `far` should be set to 0x7FFF (the largest value that can be stored in the sixteen high-order bitplanes). If there are twelve bits of z-buffer, `near` should be set to 0 and `far` should be set to 0xFFF.

```plaintext
setdepth(near, far)
Screecoord near, far;

subroutine setdep(near, far)
integer*4 near, far

procedure setdepth(near, far: Screecoord);
```

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);

2) The zclear command initializes the z-buffer to the largest possible positive integer (0x7FFF). Any point with a z coordinate less than or equal to the initial z value will be drawn to the screen.

zclear()

subroutine zclear

procedure zclear;

3) zbuffer(TRUE) turns on z-buffer; zbuffer(FALSE) turns it off.

zbuffer(bool)
Boolean bool;

subroutine zbufe(bool)
logical bool

procedure zbuffer(bool: Boolean);

getzbuffer indicates whether z-buffering is on or off. FALSE is the default value and means that z-buffering is off. TRUE means that z-buffering is on.
Boolean 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:

#include "gl.h"

main ()
{
  glinit();
  color(BLACK);
  clear();
  ortho(0.0, (float)XMAXSCREEN, 0.0, (float)YMAXSCREEN,
        0.0, -(float)XMAXSCREEN);
  setdepth(0x000, 0xFF); /* 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);
  pclose();

  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);
  pclose();

  zbuffer(FALSE); /* the IRIS exits z-buffering mode */
  gexit();
}
Figure 12.1
In Z-buffer mode, color map indices for each pixel are stored in the low-order sixteen bitplanes and the corresponding Z values for each pixel are stored in the high-order twelve or sixteen bitplanes.
12.2 Backfacing Polygon Removal

backface initiates or terminates backfacing polygon removal. A backfacing polygon is defined to be 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. Therefore, the vertices of all polygons should be specified in counterclockwise order.

The backface utility is used to improve the performance of programs which represent solid shapes as collections of polygons. The vertices of the polygons on the "far" side of the solid will be in clockwise order and will not be drawn.

Because the orientation of polygons is determined in screen space, backfacing polygon removal is not a rigorous technique for hidden surface removal. When a polygon shrinks to the point where its vertices are coincident, its orientation is indeterminant 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 previous section.)

```plaintext
backface(b)
Boolean b;

subroutine backfa(b)
  logical b

procedure backface(b: Boolean);
```

The following example draws a cube in which only the visible surfaces are drawn. The cube can be rotated by moving the mouse.

```plaintext
#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();
}
13. Shading and Depth-Cueing

The previous chapter discussed how to increase the realism of an image by removing the hidden lines and surfaces from the image. This chapter discusses two more techniques for increasing the realism of an image:

- the polygons in an image can be shaded, and
- the lines and points in an image can be depth-cued (i.e., their intensities can be varied 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 of the pixels in a filled polygon. The user controls 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 RGB value and the vertex at the other side were assigned an index with a light blue RGB value, then all of the indices between the two should be loaded with intermediate shades of blue.

The split command draws a shaded polygon. The first two arguments to split are the same as the ones in polt. (See Section 3.6.) n specifies the number of vertices in the polygon. pnts is the array of vertices. The third argument, intens, is an array of n color map indices, one for each vertex.

```plaintext
split(n, parray, iarray)
long n;
Coord parray[][][3];
Colorindex iarray[];

subroutine split(n, parray, iarray)
integer*4 n
real parray(3,n)
integer*2 iarray(n)

procedure split(n: longint; var array: Coord3array; var iarray: Colorarray);
```

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 po-
lygon 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.

Shaded polygons can also be drawn using the pmv, pdr, rpmv, and rpdr commands. (See Section 3.6.) The setshade command assigns an intensity to the vertex specified in the immediately following command. The spclos command closes and shades the polygon. If pclos is called instead, a filled, unshaded polygon is drawn in the current color and the setshade commands are ignored.

```
setshade(shade)
Colorindex shade;

subroutine setsha(shade)
integer*4 shade

procedure setshade(shade: Colorindex);
```

```
spclos()

subroutine spclos

procedure spclos;
```

The following program draws a shaded polygon using pmv, pdr, and setshade. Note that shading only works in single and double buffer display modes, and not in RGB mode.

```
#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 */
spclose();           /* close the shaded polygon */

sleep(5);
greset();
gexit();

}  

13.2 Depth-Cueing

Depth-cueing increases the perception of three dimensions in an image. When the IRIS is in depth-cue mode, the intensities of all the lines and points drawn to the screen vary according to their z values — points closer to the viewer are of higher intensity and points farther from the viewer are of lower intensity. depthcue(TRUE) turns on depth-cue mode; depthcue(FALSE) turns it off.

depthcue(mode)
Boolean mode;

subroutine depthc(mode)
logical mode

procedure depthcue(mode: Boolean);

getcue indicates whether depth-cue is on or off. TRUE means the IRIS is in depth-cue mode and FALSE means the IRIS is not in depth-cue mode.
\[ c_i = c_d + \frac{l_y - d_y}{a_y - d_y} \left[ c_a - c_d \right] \]
\[ c_r = c_c + \frac{r_y - c_y}{b_y - c_y} \left[ c_b - c_c \right] \]
\[ c_p = c_l + \frac{r_x - p_x}{r_x - l_x} \left[ c_r - c_l \right] \]

Figure 13.1
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 the intensities at point \( l \) and point \( r \).
Boolean getdcm()

logical function getdcm()

color function getdcm: longint;

The shaderange command specifies the low-intensity color map index and the high-intensity color map index. These values are mapped to the low and high $z$ values specified by $z_1$ and $z_2$. The high index must be greater than the low index and the difference between the high index and the low index must be greater than the difference between $z_1$ and $z_2$. The values of $z_1$ and $z_2$ should correspond to or lie within the range of $z$ values specifies by setdepth.

```
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: Colorindex; z1, z2: Screencoord);
```

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. The following equation yields the color map index for a point with $z$ coordinate $z$:

$$color_z = \left( \frac{lowindex - highindex}{z_2 - z_1} \right) (z - z_1) + highindex$$

Note that this equation yields a non-linear mapping when $z$ is less than $z_1$ or greater than $z_2$. Since depth-cued lines are linearly interpolated between endpoints, an endpoint outside the range of $z_1$ and $z_2$ may result in an undesirable image.
Shading and Depth-Cueing

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
#include "gl.h"
#include "device.h"
#include "math.h"

float brand();

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(brand(-200.0,200.0), brand(-200.0,200.0), brand(-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);
    movei(-200, 200, 200);
    drawi(200, 200, 200);
    drawi(200, -200, 200);
    drawi(-200, -200, 200);
    movei(200, 200, 200);
```

Version 2.1
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(0, 0x7fff);
/* 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(t);
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);
}
14. Textports

The textport is a region of the display screen used to present textual output from programs, both graphical and non-graphical. A `printf()` command in a program causes characters to be printed to the textport. `charstr`, on the other hand, 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.

The textport can be turned off with the `tpoff` command. When the textport is off, characters are not written to the textport and the textport does not appear on the screen.

```plaintext
tpoff()

subroutine tpoff

procedure tpoff;
```

The textport will be automatically turned on when the program in which the `tpoff` command was called is exited. The textport can be turned on explicitly with the `tpon` command.

```plaintext
tpon()

subroutine tpone

procedure tpone;
```

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(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: Screencoord);

textport(0,0,0,0) turns off the textport and it will not appear on the screen until
the textport size is reset or textlimit (see below) is called. The gettp command re-
turns 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);

The color of the text in a textport is set with the textcolor command. Note that
the color of text written in the viewport with charstr is determined by color.

textcolor(tcolor)
Colorindex tcolor;

subroutine textco(tcolor)
integer*4 tcolor

procedure textcolor(tcolor: Colorindex);

The textwritemask command grants write permission for text drawn in the
textport. This command does not affect text drawn with charstr.
textwritemask(tmask)
Colorindex tmask;

subroutine textwr(tmask)
integer*4 tmask

procedure textwritemask(tmask: Colorindex);

The pagecolor command sets the color for the background of the textport.

pagecolor(c)
short c;

subroutine pageco(c)
integer*4 c

procedure pagecolor(c: Colorindex);

The pagewritemask command sets the writemask for the background of the textport.

pagewritemask(pmask)
short pmask;

subroutine pagewr(pmask)
integer*4 pmask

procedure pagewritemask(pmask: Colorindex);

The textport can be reset to its default values with the textinit command.
textinit()

subroutine textin

procedure textinit;
Appendix A
Type Definitions for C, FORTRAN, and Pascal

A.1 C Definitions
This is a listing of "gl.h", which should be included in each IRIS program. It contains the type definitions, useful constants, and external definitions for all commands.

/* graphics library header file */

/* maximum X and Y screen coordinates */

#define XMAXSCREEN  1023
#define YMAXSCREEN  767

/* various hardware/software limits */

#define ATTRIBSTACKDEPTH  10
#define VPSTACKDEPTH  8
#define MATRIXSTACKDEPTH  32
#define NAMESTACKDEPTH  1025
#define STARTTAG  -2
#define ENDTAG  -3

/* names for colors in color map loaded by ginit() */

#define BLACK  0
#define RED  1
#define GREEN  2
#define YELLOW  3
#define BLUE  4
#define MAGENTA  5
#define CYAN  6
#define WHITE  7

#ifndef FALSE
#define FALSE
#endif
#ifndef TRUE
#define TRUE /*!FALSE)
#endif

Version 2.1

IRIS User's Guide
/* typedef */

typedef unsigned char Byte;
typedef long Boolean;
typedef char *String;

typedef short Angle;
typedef short Screeencoord;
typedef short Scoord;
typedef long Icoord;
typedef float Coord;
typedef float Matrix[4][4];

typedef unsigned short Colorindex;
typedef unsigned char RGBvalue;

typedef unsigned short Device;

#define PATTERN_16 16
#define PATTERN_32 32
#define PATTERN_64 64

#define PATTERN_16_SIZE 16
#define PATTERN_32_SIZE 64
#define PATTERN_64_SIZE 256

typedef unsigned short Pattern16[PATTERN_16_SIZE];
typedef unsigned short Pattern32[PATTERN_32_SIZE];
typedef unsigned short Pattern64[PATTERN_64_SIZE];

typedef unsigned short Linestyle;
typedef unsigned short Cursor[16];
typedef struct {
    unsigned short offset;
    Byte w,h;
    char xoff,yoff;
    short width;
} Fontchar;

typedef long Object;
typedef long Tag;
typedef long Offset;
extern void arc();
extern void arcf();
extern void arcfi();
extern void arci();
exern void arcf();
exern void arcs();
exern void attachcursor();
exern void backbuffer();
exern void backface();
exern void bbox2();
exern void bbox2i();
exern void bbox2s();
exern void blankscreen();
exern void blink();
exern long blkqread();
exern void callfunc();
exern void calloc();
exern void charstr();
exern void circ();
exern void circf();
exern void circf1();
exern void circi();
exern void circfs();
exern void circs();
exern void clear();
exern void clearbitcode();
exern void clkoff();
exern void clkon();
exern void closeobj();
exern void cmov();
exern void cmov2();
exern void cmov2i();
exern void cmovi();
exern void cmov2s();
exern void cmovs();
exern void color();
exern void compactify();
exern void crv();
exern void crvn();
exern void curorigin();
exern void curoff();
exern void curson();
exern void curvebasis();
exern void curveit();
exern void curveprecision();
exern void cyclemap();
exern void defbasis();
exern void defcursor();
extern void    deflinestyle();
extern void    defpattern();
extern void    defrasterfont();
extern void    delobj();
extern void    deltag();
extern void    depthcue();
extern void    doublebuffer();
extern void    draw();
extern void    draw2();
extern void    draw2i();
extern void    drawi();
extern void    draw2s();
extern void    draws();
extern void    edithobj();
extern long    endfeedback();
extern long    endpick();
extern long    endselect();
extern void    feedback();
extern void    finish();
extern void    font();
extern void    frontbuffer();
extern void    gconfig();
extern Object  genobj();
extern Tag     gentag();
extern long    getbuffer();
extern long    getbutton();
extern long    getcmmode();
extern long    getcolor();
extern void    getcpos();
extern void    getcursor();
extern long    getdc();
extern void    getdepth();
extern long    getdisplaymode();
extern long    getfont();
extern void    getgpos();
extern long    getheight();
extern long    gethitcode();
extern long    getsbackup();
extern long    getsrepeat();
extern long    getstyle();
extern long    getwidth();
extern long    getmap();
extern void    getmatrix();
extern void    getmcolor();
extern long    getmem();
extern long getmonitor();
extern Object getopenobj();
extern long getpattern();
extern long getplanes();
extern long getresetls();
extern void getscreenmask();
extern long getvaluator();
extern void getviewport();
extern long getwritemask();
extern long getzbuffer();
extern void gwrite();
extern void gexit();
extern void ginit();
extern void greset();
extern void gsync();
extern void gRGBcolor();
extern void gRGBcursor();
extern void gRGBmask();
extern void initnames();
extern long isobj();
extern long istag();
extern void lampoff();
extern void lampon();
extern void linewidth();
extern void loadmatrix();
extern void loadname();
extern void lookat();
extern void lsbackup();
extern void lsrepeat();
extern void makeobj();
extern void maketag();
extern void mapcolor();
extern void mapw();
extern void mapw2();
extern void move();
extern void move2();
extern void move2i();
extern void movei();
extern void move2s();
extern void moves();
extern void multimap();
extern void multmatrix();
extern void newtag();
extern void noise();
extern void objdelete();
extern void objinsert();
extern void objreplace();
extern void onemap();
extern void ortho();
extern void ortho2();
extern void pagecolor();
extern void pagewritemask();
extern void passthrough();
extern void patch();
extern void patchbasis();
extern void patchprecision();
extern void pclos();
extern void spclos();
extern void pdr();
extern void pdr2();
extern void pdr2i();
extern void pdr1();
extern void pdr2s();
extern void pdrs();
extern void perspective();
extern void pick();
extern void picksiz();
extern void pmv();
extern void pmv2();
extern void pmv2i();
extern void pmvi();
extern void pmv2s();
extern void pmvs();
extern void pnt();
extern void pnt2();
extern void pnt2i();
extern void pnti();
extern void pnt2s();
extern void pnts();
extern void polarview();
extern void polif();
extern void polif2();
extern void polif2i();
extern void polifi();
extern void polif2s();
extern void polifs();
extern void poly();
extern void poly2();
extern void poly2i();
extern void polyi();
extern void poly2s();
extern void polys();
extern void popattributes();
extern void popmatrix();
extern void popname();
extern void popviewport();
extern void pushattributes();
extern void pushmatrix();
extern void pushname();
extern void pushviewport();
extern void qdevice();
extern void qcenter();
extern void qread();
extern void qreset();
extern void qtest();
extern void rdr();
extern void rdr2();
extern void rdr2i();
extern void rdri();
extern void rdr2s();
extern void drs();
extern void readpixels();
extern void readRGB();
extern void rect();
extern void rectf();
extern void rectfi();
extern void recti();
extern void rectfs();
extern void rects();
extern void resets();
extern void RGBcolor();
extern void RGBcursor();
extern void RGBmode();
extern void RGBwritemask();
extern void ringbell();
extern void rmv();
extern void rmv2();
extern void rmv2i();
extern void rmvi();
extern void rmv2s();
extern void rmvs();
extern void rotate();
extern void rpdr();
extern void rpdr2();
extern void rpdr2i();
 extern void rpdri();
 extern void rpdr2s();
 extern void rpdras();
 extern void rpsmv();
 extern void rpsmv2();
 extern void rpsmv2i();
 extern void rpsvi();
 extern void rpsv2s();
 extern void rpsvs();
 extern void scale();
 extern void scrmask();
 extern void select();
 extern void setbell();
 extern void setbutton();
 extern void setcursor();
 extern void setdepth();
 extern void setlinestyle();
 extern void setmap();
 extern void setmonitor();
 extern void setpattern();
 extern void setshade();
 extern void setvaluator();
 extern void shaderange();
 extern void singlebuffer();
 extern void splf();
 extern void splf2();
 extern void splf2i();
 extern void splfi();
 extern void splf2s();
 extern void splfs();
 extern long strwidth();
 extern void swapbuffers();
 extern void swapinterval();
 extern void textcolor();
 extern void textport();
 extern void textwritemask();
 extern void tie();
 extern void tpoft();
 extern void tpon();
 extern void translate();
 extern void viewport();
 extern void window();
 extern void writemask();
 extern void writepixels();
 extern void writeRGB();
extern void       xfpt();
extern void       xfpt2();
extern void       xfpt2i();
extern void       xfpti();
extern void       xfpts();
extern void       xfpt2s();
extern void       xfpt4();
extern void       xfpt4i();
extern void       xfpt4s();
extern void       zbuffer();
extern void       zclear();
Another header file, "device.h", defines symbolic names for many of the device numbers. IRIS programmers are not required to use it. In fact, they may wish to define their own names for the subset of devices actually used in a particular application program.

/* macros to test valuator and button numbers */

#define ISBUTTON(b)    (((b) >= BUTOFFSET) && ((b) < (BUTCOUNT + BUTOFFSET)) )
#define ISSCBUTTON(b)   (((b) >= SBTOFFSET) && ((b) < (SBTCOUNT + SBTOFFSET)) )
#define ISVALUATOR(v)   (((v) >= VALOFFSET) && ((v) < (VALCOUNT + VALOFFSET)) )
#define ISTIWER(t)      (((t) >= TIMOFFSET) && ((t) < (TIMCOUNT + TIMOFFSET)) )
#define ISDIAL(t)       (((t) >= DIAL0) && ((t) < DIAL8))
#define ISLPEN(t)       (((t) == LPEX) || ((t) == LPEY))
#define ISLPENBUT(t)    ((t) == LPEBUT)
#define ISBPADBUT(t)    (((t) >= BPADO) && ((t) < BPADO))

/* include file with device definitions */
#define NULLDEV 0
#define BUTOFFSET 1
#define SBTOFFSET 200
#define VALOFFSET 256
#define KEYOFFSET 512
#define TIMOFFSET 515
#define BUTCOUNT 144
#define SBTCOUNT 16
#define VALCOUNT 20
#define TIMCOUNT 8

/* buttons */

#define BUT0 1 /* 0+BUTOFFSET */
#define BUT1 2 /* 1+BUTOFFSET */
#define BUT2 3 /* 2+BUTOFFSET */
#define BUT3 4 /* 3+BUTOFFSET */
#define BUT4 5 /* 4+BUTOFFSET */
#define BUT5 6 /* 5+BUTOFFSET */
#define BUT6 7 /* 6+BUTOFFSET */
#define BUT7 8 /* 7+BUTOFFSET */
#define BUT8 9 /* 8+BUTOFFSET */
#define BUT9 10 /* 9+BUTOFFSET */
#define BUT10 11 /* 10+BUTOFFSET, A */
#define BUT11 12 /* 11+BUTOFFSET, B */
#define BUT12 13 /* 12+BUTOFFSET, C */
#define BUT13 14 /* 13+BUTOFFSET, D */
#define BUT14 15 /* 14+BUTOFFSET, E */
| #define BUT15 | 16 | /* 15+BUTOFFSET, F */ |
| #define BUT16 | 17 | /* 16+BUTOFFSET, 10 */ |
| #define BUT17 | 18 | /* 17+BUTOFFSET, 11 */ |
| #define BUT18 | 19 | /* 18+BUTOFFSET, 12 */ |
| #define BUT19 | 20 | /* 19+BUTOFFSET, 13 */ |
| #define BUT20 | 21 | /* 20+BUTOFFSET, 14 */ |
| #define BUT21 | 22 | /* 21+BUTOFFSET, 15 */ |
| #define BUT22 | 23 | /* 22+BUTOFFSET, 16 */ |
| #define BUT23 | 24 | /* 23+BUTOFFSET, 17 */ |
| #define BUT24 | 25 | /* 24+BUTOFFSET, 18 */ |
| #define BUT25 | 26 | /* 25+BUTOFFSET, 19 */ |
| #define BUT26 | 27 | /* 26+BUTOFFSET, 1A */ |
| #define BUT27 | 28 | /* 27+BUTOFFSET, 1B */ |
| #define BUT28 | 29 | /* 28+BUTOFFSET, 1C */ |
| #define BUT29 | 30 | /* 29+BUTOFFSET, 1D */ |
| #define BUT30 | 31 | /* 30+BUTOFFSET, 1E */ |
| #define BUT31 | 32 | /* 31+BUTOFFSET, 1F */ |
| #define BUT32 | 33 | /* 32+BUTOFFSET, 20 */ |
| #define BUT33 | 34 | /* 33+BUTOFFSET, 21 */ |
| #define BUT34 | 35 | /* 34+BUTOFFSET, 22 */ |
| #define BUT35 | 36 | /* 35+BUTOFFSET, 23 */ |
| #define BUT36 | 37 | /* 36+BUTOFFSET, 24 */ |
| #define BUT37 | 38 | /* 37+BUTOFFSET, 25 */ |
| #define BUT38 | 39 | /* 38+BUTOFFSET, 26 */ |
| #define BUT39 | 40 | /* 39+BUTOFFSET, 27 */ |
| #define BUT40 | 41 | /* 40+BUTOFFSET, 28 */ |
| #define BUT41 | 42 | /* 41+BUTOFFSET, 29 */ |
| #define BUT42 | 43 | /* 42+BUTOFFSET, 2A */ |
| #define BUT43 | 44 | /* 43+BUTOFFSET, 2B */ |
| #define BUT44 | 45 | /* 44+BUTOFFSET, 2C */ |
| #define BUT45 | 46 | /* 45+BUTOFFSET, 2D */ |
| #define BUT46 | 47 | /* 46+BUTOFFSET, 2E */ |
| #define BUT47 | 48 | /* 47+BUTOFFSET, 2F */ |
| #define BUT48 | 49 | /* 48+BUTOFFSET, 30 */ |
| #define BUT49 | 50 | /* 49+BUTOFFSET, 31 */ |
| #define BUT50 | 51 | /* 50+BUTOFFSET, 32 */ |
| #define BUT51 | 52 | /* 51+BUTOFFSET, 33 */ |
| #define BUT52 | 53 | /* 52+BUTOFFSET, 34 */ |
| #define BUT53 | 54 | /* 53+BUTOFFSET, 35 */ |
| #define BUT54 | 55 | /* 54+BUTOFFSET, 36 */ |
| #define BUT55 | 56 | /* 55+BUTOFFSET, 37 */ |
| #define BUT56 | 57 | /* 56+BUTOFFSET, 38 */ |
| #define BUT57 | 58 | /* 57+BUTOFFSET, 39 */ |
| #define BUT58 | 59 | /* 58+BUTOFFSET, 3A */ |
| #define BUT59 | 60 | /* 59+BUTOFFSET, 3B */ |
#define BUT128 129 /* 128+BUTOFFSET */
#define BUT129 130 /* 129+BUTOFFSET */
#define BUT130 131 /* 130+BUTOFFSET */
#define BUT131 132 /* 131+BUTOFFSET */
#define BUT132 133 /* 132+BUTOFFSET */
#define BUT133 134 /* 133+BUTOFFSET */
#define BUT134 135 /* 134+BUTOFFSET */
#define BUT135 136 /* 135+BUTOFFSET */
#define BUT136 137 /* 136+BUTOFFSET */
#define BUT137 138 /* 137+BUTOFFSET */
#define BUT138 139 /* 138+BUTOFFSET */
#define BUT139 140 /* 139+BUTOFFSET */
#define BUT140 141 /* 140+BUTOFFSET */
#define BUT141 142 /* 141+BUTOFFSET */
#define MOUSE1 101 /* BUT100 */
#define MOUSE2 102 /* BUT101 */
#define MOUSE3 103 /* BUT102 */
#define LEFTMOUSE 103 /* BUT102 */
#define MIDDLEMOUSE 102 /* BUT101 */
#define RIGHTMOUSE 101 /* BUT100 */
#define LPENBUT 104 /* LIGHT PEN BUTTON */
#define BPAD0 105 /* BITPAD BUTTON 0 */
#define BPAD1 106 /* BITPAD BUTTON 1 */
#define BPAD2 107 /* BITPAD BUTTON 2 */
#define BPAD3 108 /* BITPAD BUTTON 3 */
#define SWBASE 111 /* BUT110 */
#define SW0 111 /* SWBASE */
#define SW1 112 /* SWBASE+1 */
#define SW2 113 /* SWBASE+2 */
#define SW3 114 /* SWBASE+3 */
#define SW4 115 /* SWBASE+4 */
#define SW5 116 /* SWBASE+5 */
#define SW6 117 /* SWBASE+6 */
#define SW7 118 /* SWBASE+7 */
#define SW8 119 /* SWBASE+8 */
#define SW9 120 /* SWBASE+9 */
#define SW10 121 /* SWBASE+10 */
#define SW11 122 /* SWBASE+11 */
#define SW12 123 /* SWBASE+12 */
#define SW13 124 /* SWBASE+13 */
#define SW14 125 /* SWBASE+14 */
#define SW15 126 /* SWBASE+15 */
#define SW16 127 /* SWBASE+16 */
#define SW17 128 /* SWBASE+17 */
#define SW18 129 /* SWBASE+18 */
#define SW19 130 /* SWBASE+19 */
#define SW20 131 /* SWBASE+20 */
#define SW21 132 /* SWBASE+21 */
#define SW22 133 /* SWBASE+22 */
#define SW23 134 /* SWBASE+23 */
#define SW24 135 /* SWBASE+24 */
#define SW25 136 /* SWBASE+25 */
#define SW26 137 /* SWBASE+26 */
#define SW27 138 /* SWBASE+27 */
#define SW28 139 /* SWBASE+28 */
#define SW29 140 /* SWBASE+29 */
#define SW30 141 /* SWBASE+30 */
#define SW31 142 /* SWBASE+31 */

#define AKEY 11 /* BUT10 */
#define BKEY 36 /* BUT35 */
#define CKEY 28 /* BUT27 */
#define DKEY 18 /* BUT17 */
#define EKEY 17 /* BUT16 */
#define FKEY 19 /* BUT18 */
#define GKEY 26 /* BUT25 */
#define HKEY 27 /* BUT26 */
#define IKEY 40 /* BUT39 */
#define JKEY 34 /* BUT33 */
#define KKEY 35 /* BUT34 */
#define LKEY 42 /* BUT41 */
#define MKEY 44 /* BUT43 */
#define NKEY 37 /* BUT36 */
#define OKEY 41 /* BUT40 */
#define PKEY 48 /* BUT47 */
#define QKEY 10 /* BUT9 */
#define RKEY 24 /* BUT23 */
#define SKEY 12 /* BUT11 */
#define TKEY 25 /* BUT24 */
#define UKEY 33 /* BUT32 */
#define VKEY 29 /* BUT28 */
#define WKEY 16 /* BUT15 */
#define XKEY 21 /* BUT20 */
#define YKEY 32 /* BUT31 */
#define ZKEY 20 /* BUT19 */
#define ZEROKEY 46 /* BUT45 */
#define ONEKEY 8 /* BUT7 */
#define TWOKEY 14 /* BUT13 */
#define THREEKEY 15 /* BUT14 */
<table>
<thead>
<tr>
<th>#define</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOURKEY</td>
<td>22</td>
<td>/* BUT21 */</td>
</tr>
<tr>
<td>FIVEKEY</td>
<td>23</td>
<td>/* BUT22 */</td>
</tr>
<tr>
<td>SIXKEY</td>
<td>30</td>
<td>/* BUT29 */</td>
</tr>
<tr>
<td>SEVENKEY</td>
<td>31</td>
<td>/* BUT30 */</td>
</tr>
<tr>
<td>EIGHTKEY</td>
<td>38</td>
<td>/* BUT37 */</td>
</tr>
<tr>
<td>NINEKEY</td>
<td>39</td>
<td>/* BUT38 */</td>
</tr>
<tr>
<td>BREAKKEY</td>
<td>1</td>
<td>/* BUT0 */</td>
</tr>
<tr>
<td>SETUPKEY</td>
<td>2</td>
<td>/* BUT1 */</td>
</tr>
<tr>
<td>CTRLKEY</td>
<td>3</td>
<td>/* BUT2 */</td>
</tr>
<tr>
<td>CAPSLOCKKEY</td>
<td>4</td>
<td>/* BUT3 */</td>
</tr>
<tr>
<td>RIGHTSHIFTKEY</td>
<td>5</td>
<td>/* BUT4 */</td>
</tr>
<tr>
<td>LEFTSHIFTKEY</td>
<td>6</td>
<td>/* BUT5 */</td>
</tr>
<tr>
<td>NOSCRLKEY</td>
<td>13</td>
<td>/* BUT12 */</td>
</tr>
<tr>
<td>ESCKEY</td>
<td>7</td>
<td>/* BUT6 */</td>
</tr>
<tr>
<td>TABKEY</td>
<td>9</td>
<td>/* BUT8 */</td>
</tr>
<tr>
<td>RETKEY</td>
<td>51</td>
<td>/* BUT50 */</td>
</tr>
<tr>
<td>SPACEKEY</td>
<td>83</td>
<td>/* BUT82 */</td>
</tr>
<tr>
<td>LINEFEEDKEY</td>
<td>60</td>
<td>/* BUT59 */</td>
</tr>
<tr>
<td>BACKSPACEKEY</td>
<td>61</td>
<td>/* BUT60 */</td>
</tr>
<tr>
<td>DELKEY</td>
<td>62</td>
<td>/* BUT61 */</td>
</tr>
<tr>
<td>SEMICOLONKEY</td>
<td>43</td>
<td>/* BUT42 */</td>
</tr>
<tr>
<td>PERIODKEY</td>
<td>52</td>
<td>/* BUT51 */</td>
</tr>
<tr>
<td>COMMAKEY</td>
<td>45</td>
<td>/* BUT44 */</td>
</tr>
<tr>
<td>QUOTEKEY</td>
<td>50</td>
<td>/* BUT49 */</td>
</tr>
<tr>
<td>ACCENTGRAVEKEY</td>
<td>55</td>
<td>/* BUT54 */</td>
</tr>
<tr>
<td>MINUSKEY</td>
<td>47</td>
<td>/* BUT46 */</td>
</tr>
<tr>
<td>VIRGULEKEY</td>
<td>53</td>
<td>/* BUT52 */</td>
</tr>
<tr>
<td>BACKSLASHKEY</td>
<td>57</td>
<td>/* BUT56 */</td>
</tr>
<tr>
<td>EQUALKEY</td>
<td>54</td>
<td>/* BUT53 */</td>
</tr>
<tr>
<td>LEFTBRACKETKEY</td>
<td>49</td>
<td>/* BUT48 */</td>
</tr>
<tr>
<td>RIGHTBRACKETKEY</td>
<td>56</td>
<td>/* BUT55 */</td>
</tr>
<tr>
<td>LEFTARROWKEY</td>
<td>73</td>
<td>/* BUT72 */</td>
</tr>
<tr>
<td>DOWNARROWKEY</td>
<td>74</td>
<td>/* BUT73 */</td>
</tr>
<tr>
<td>RIGHTARROWKEY</td>
<td>80</td>
<td>/* BUT79 */</td>
</tr>
<tr>
<td>UPARROWKEY</td>
<td>81</td>
<td>/* BUT80 */</td>
</tr>
<tr>
<td>PAD0</td>
<td>59</td>
<td>/* BUT58 */</td>
</tr>
<tr>
<td>PAD1</td>
<td>58</td>
<td>/* BUT57 */</td>
</tr>
<tr>
<td>PAD2</td>
<td>64</td>
<td>/* BUT63 */</td>
</tr>
<tr>
<td>PAD3</td>
<td>65</td>
<td>/* BUT64 */</td>
</tr>
<tr>
<td>PAD4</td>
<td>63</td>
<td>/* BUT62 */</td>
</tr>
<tr>
<td>PAD5</td>
<td>69</td>
<td>/* BUT68 */</td>
</tr>
<tr>
<td>PAD6</td>
<td>70</td>
<td>/* BUT69 */</td>
</tr>
<tr>
<td>PAD7</td>
<td>67</td>
<td>/* BUT66 */</td>
</tr>
<tr>
<td>PAD8</td>
<td>68</td>
<td>/* BUT67 */</td>
</tr>
<tr>
<td>PAD9</td>
<td>75</td>
<td>/* BUT74 */</td>
</tr>
</tbody>
</table>
#define PADFF1 72 /* BUT71 */
#define PADFF2 71 /* BUT70 */
#define PADFF3 79 /* BUT78 */
#define PADFF4 78 /* BUT77 */
#define PADPERIOD 66 /* BUT65 */
#define PADMINUS 76 /* BUT75 */
#define PADCOMMA 77 /* BUT76 */
#define PADERENTER 82 /* BUT81 */

/* screen buttons */

#define SCRBUF0 200 /* 0+SBTOFFSET */
#define SCRBUF1 201 /* 1+SBTOFFSET */
#define SCRBUF2 202 /* 2+SBTOFFSET */
#define SCRBUF3 203 /* 3+SBTOFFSET */
#define SCRBUF4 204 /* 4+SBTOFFSET */
#define SCRBUF5 205 /* 5+SBTOFFSET */
#define SCRBUF6 206 /* 6+SBTOFFSET */
#define SCRBUF7 207 /* 7+SBTOFFSET */
#define SCRBUF8 208 /* 8+SBTOFFSET */
#define SCRBUF9 209 /* 9+SBTOFFSET */
#define SCRBUF10 210 /* 10+SBTOFFSET */
#define SCRBUF11 211 /* 11+SBTOFFSET */
#define SCRBUF12 212 /* 12+SBTOFFSET */
#define SCRBUF13 213 /* 13+SBTOFFSET */
#define SCRBUF14 214 /* 14+SBTOFFSET */
#define SCRBUF15 215 /* 15+SBTOFFSET */
#define SCRBUF16 216 /* 16+SBTOFFSET */

/* valuators */

#define SGIRESERVED 256 /* 0+VALOFFSET */
#define DIAL0 257 /* 1+VALOFFSET */
#define DIAL1 258 /* 2+VALOFFSET */
#define DIAL2 259 /* 3+VALOFFSET */
#define DIAL3 260 /* 4+VALOFFSET */
#define DIAL4 261 /* 5+VALOFFSET */
#define DIAL5 262 /* 6+VALOFFSET */
#define DIAL6 263 /* 7+VALOFFSET */
#define DIAL7 264 /* 8+VALOFFSET */
#define DIAL8 265 /* 9+VALOFFSET */
#define MOUSEX 266 /* 10+VALOFFSET */
#define MOUSEY 267 /* 11+VALOFFSET */
#define LPENX 268 /* 12+VALOFFSET */
#define LPENY 269 /* 13+VALOFFSET */
/* timers */
#define TIMER0 515 /* 0*TIMOFFSET */
#define TIMER1 516 /* 1*TIMOFFSET */
#define TIMER2 517 /* 2*TIMOFFSET */
#define TIMER3 518 /* 3*TIMOFFSET */
#define TIMER4 519 /* 4*TIMOFFSET */
#define TIMER5 520 /* 5*TIMOFFSET */
#define TIMER6 521 /* 6*TIMOFFSET */
#define TIMER7 522 /* 7*TIMOFFSET */

/* misc devices */
#define KEYBD 513 /* keyboard */
#define RAWKEYBD 514 /* raw keyboard for keyboard manager */
#define VALMARK 523 /* valuator mark */
#define GERROR 524 /* errors device */
#define REDRAW 528 /* used by port manager to signal redraws */
#define WMSEND 529 /* data in proc's shmem */
#define WMREPLY 530 /* reply from port manager */
#define WMFCLOSE 531 /* gf # is no longer being used */
#define WMXCLOSE 532 /* tx # is no longer being used */
#define MODECHANGE 533 /* the display mode has changed */
#define INPUTCHANGE 534 /* input connected or disconnected */
#define QFULL 535 /* queue was filled */
#define PIECECHANGE 536 /* change in the window pieces */
A third header file, "get.h", defines constants for the values of global attributes like display mode and update buffers. It too is optional.

/* include file containing definitions for returned values of get* routines */

/* values returned by getbuffer() */

#define NOBUFFER 0
#define BCKBUFFER 1
#define FRNTBUFFER 2
#define BOTHBUFFERS 3

/* values returned by getcmmode() */

#define CMAPMULTI 0
#define CMAFONE 1

/* values returned by getdisplaymode() */

#define DMRGB 0
#define DMSINGLE 1
#define DMDOUBLE 2

/* values returned by getmonitor() */

#define HZ30 0
#define HZ60 1
#define NTSC 2
#define PAL 2
#define HZ50 3
#define MONA 5
#define MONB 6
#define MONC 7
#define MOND 8
#define MONSECIAL 0x20

/* individual hit bits returned by gethitcode() */

#define FARPLANE 0x0001
#define NEARPLANE 0x0002
#define TOPPLANE 0x0004
#define BOTTOMPLANE 0x0008
#define RIGHTPLANE 0x0010
#define LEFTPLANE 0x0020
A.2 FORTRAN Definitions

C "fg1.h"

C graphics library header file

C maximum X and Y screen coordinates

INTEGER*4 XMAXSC
INTEGER*4 YMAXSC

C various hardware/software limits

INTEGER*4 ATTRIB
INTEGER*4 VPSTAC
INTEGER*4 MATRIX
INTEGER*4 STARTT
INTEGER*4 ENDTAG

C names for colors in color map loaded by ginit()

INTEGER*4 BLACK
INTEGER*4 RED
INTEGER*4 GREEN
INTEGER*4 YELLOW
INTEGER*4 BLUE
INTEGER*4 MAGENT
INTEGER*4 CYAN
INTEGER*4 WHITE

C function return values

INTEGER*4 BLKQRE
INTEGER*4 ENDFEE
INTEGER*4 ENDFIC
INTEGER*4 ENDSEL
INTEGER*4 GENOBJ
INTEGER*4 GENTAG
INTEGER*4 GETBUF
LOGICAL GETBUT
LOGICAL GETCM
INTEGER*4 GETCOL
LOGICAL GETDCM
INTEGER*4 GETDIS
INTEGER*4 GETFON
INTEGER*4    GETHEI
INTEGER*4    GETHIT
LOGICAL      GETLSB
INTEGER*4    GETLSR
INTEGER*4    GETLST
INTEGER*4    GETLWI
INTEGER*4    GETMAP
INTEGER*4    GETMEM
INTEGER*4    GETMON
INTEGER*4    GETOPE
INTEGER*4    GETPAT
INTEGER*4    GETPLA
LOGICAL      GETRES
INTEGER*4    GETVAL
INTEGER*4    GETWRI
LOGICAL      GETZBU
INTEGER*4    GVERS1
LOGICAL      ISOBJ
LOGICAL      ISTAT
INTEGER*4    QREAD
INTEGER*4    QTEST
INTEGER*4    READPI
INTEGER*4    READRG
INTEGER*4    STRWID

C  maximum X and Y screen coordinates

PARAMETER (XMAXSC = 1023)
PARAMETER (YMAXSC = 767)

C  various hardware/software limits

PARAMETER (ATTRIB = 10)
PARAMETER (VPSTAC = 8)
PARAMETER (MATRIX = 32)
PARAMETER (STARTT = -2)
PARAMETER (ENDTAG = -3)

C  names for colors in color map loaded by ginit()

PARAMETER (BLACK = 0)
PARAMETER (RED = 1)
PARAMETER (GREEN = 2)
PARAMETER (YELLOW= 3)
PARAMETER (BLUE = 4)
PARAMETER (MAGENT= 5)
PARAMETER (CYAN =  6)
PARAMETER (WHITE =  7)

C  "fget.h"

integer*4 NOBUFF
integer*4 BCKBUF
integer*4 FRMTBU
integer*4 BOTHBU
integer*4 CMAPMU
integer*4 CMAPON
integer*4 DMRGB
integer*4 DMSING
integer*4 DMDOUB
integer*4 Hz30
integer*4 Hz60
integer*4 NTSC
integer*4 PAL
integer*4 Hz50
integer*4 MONA
integer*4 MONB
integer*4 MONC
integer*4 MOND
integer*4 MONSPE
integer*4 FARPLA
integer*4 NEARPL
integer*4 TOPPLA
integer*4 BOTTOM
integer*4 RIGHTP
integer*4 LEFTPL

C include file containing definitions for returned values of get* routines

c values returned by getbuffer()

PARAMETER ( NOBUFF =  0 )
PARAMETER ( BCKBUF =  1 )
PARAMETER ( FRMTBU =  2 )
PARAMETER ( BOTHBU =  3 )
c values returned by getcmmode()

PARAMETER ( CMAPMU = 0 )
PARAMETER ( CMAPON = 1 )

c values returned by getdisplaymode()

PARAMETER ( DMRGB = 0 )
PARAMETER ( DMSING = 1 )
PARAMETER ( DMDOUB = 2 )

C values returned by getmonitor()

PARAMETER ( HZ30 = 0 )
PARAMETER ( HZ60 = 1 )
PARAMETER ( NTSC = 2 )
PARAMETER ( PAL = 2 )
PARAMETER ( HZ50 = 3 )
PARAMETER ( MONA = 5 )
PARAMETER ( MONB = 6 )
PARAMETER ( MONC = 7 )
PARAMETER ( MOND = 8 )
PARAMETER ( MONSPE = 32 )

C individual hit bits returned by gethitcode()

PARAMETER ( FARPLA = 1 )
PARAMETER ( WEARPL = 2 )
PARAMETER ( TOPPLA = 4 )
PARAMETER ( BOTTOM = 8 )
PARAMETER ( RIGHTP = 16 )
PARAMETER ( LEFTPL = 32 )

C "fdevice.h"
C include file with device definitions

INTEGER*4 NULLDE
INTEGER*4 BUTOFF
INTEGER*4 SBTOFF
INTEGER*4 VALOFF
INTEGER*4 KEYOFF
INTEGER*4 TIMOFF
INTEGER*4 BUTCOU
INTEGER*4 SBTCOU
INTEGER*4 VALCOU
INTEGER*4 TIMCOU

C buttons

INTEGER*4 BUT0
INTEGER*4 BUT1
INTEGER*4 BUT2
INTEGER*4 BUT3
INTEGER*4 BUT4
INTEGER*4 BUT5
INTEGER*4 BUT6
INTEGER*4 BUT7
INTEGER*4 BUT8
INTEGER*4 BUT9
INTEGER*4 BUT10
INTEGER*4 BUT11
INTEGER*4 BUT12
INTEGER*4 BUT13
INTEGER*4 BUT14
INTEGER*4 BUT15
INTEGER*4 BUT16
INTEGER*4 BUT17
INTEGER*4 BUT18
INTEGER*4 BUT19
INTEGER*4 BUT20
INTEGER*4 BUT21
INTEGER*4 BUT22
INTEGER*4 BUT23
INTEGER*4 BUT24
INTEGER*4 BUT25
INTEGER*4 BUT26
INTEGER*4 BUT27
INTEGER*4 BUT28
INTEGER*4 BUT29
INTEGER*4 BUT30
INTEGER*4 BUT31
INTEGER*4 BUT32
INTEGER*4 BUT33
INTEGER*4 BUT34
INTEGER*4 BUT35
INTEGER*4 BUT36
INTEGER 4 BUT37
INTEGER 4 BUT38
INTEGER 4 BUT39
INTEGER 4 BUT40
INTEGER 4 BUT41
INTEGER 4 BUT42
INTEGER 4 BUT43
INTEGER 4 BUT44
INTEGER 4 BUT45
INTEGER 4 BUT46
INTEGER 4 BUT47
INTEGER 4 BUT48
INTEGER 4 BUT49
INTEGER 4 BUT50
INTEGER 4 BUT51
INTEGER 4 BUT52
INTEGER 4 BUT53
INTEGER 4 BUT54
INTEGER 4 BUT55
INTEGER 4 BUT56
INTEGER 4 BUT57
INTEGER 4 BUT58
INTEGER 4 BUT59
INTEGER 4 BUT60
INTEGER 4 BUT61
INTEGER 4 BUT62
INTEGER 4 BUT63
INTEGER 4 BUT64
INTEGER 4 BUT65
INTEGER 4 BUT66
INTEGER 4 BUT67
INTEGER 4 BUT68
INTEGER 4 BUT69
INTEGER 4 BUT70
INTEGER 4 BUT71
INTEGER 4 BUT72
INTEGER 4 BUT73
INTEGER 4 BUT74
INTEGER 4 BUT75
INTEGER 4 BUT76
INTEGER 4 BUT77
INTEGER 4 BUT78
INTEGER 4 BUT79
INTEGER 4 BUT80
INTEGER 4 BUT81
INTEGER*4 BUT82
INTEGER*4 MAXKBD
INTEGER*4 BUT100
INTEGER*4 BUT101
INTEGER*4 BUT102
INTEGER*4 BUT110
INTEGER*4 BUT111
INTEGER*4 BUT112
INTEGER*4 BUT113
INTEGER*4 BUT114
INTEGER*4 BUT115
INTEGER*4 BUT116
INTEGER*4 BUT117
INTEGER*4 BUT118
INTEGER*4 BUT119
INTEGER*4 BUT120
INTEGER*4 BUT121
INTEGER*4 BUT122
INTEGER*4 BUT123
INTEGER*4 BUT124
INTEGER*4 BUT125
INTEGER*4 BUT126
INTEGER*4 BUT127
INTEGER*4 BUT128
INTEGER*4 BUT129
INTEGER*4 BUT130
INTEGER*4 BUT131
INTEGER*4 BUT132
INTEGER*4 BUT133
INTEGER*4 BUT134
INTEGER*4 BUT135
INTEGER*4 BUT136
INTEGER*4 BUT137
INTEGER*4 BUT138
INTEGER*4 BUT139
INTEGER*4 BUT140
INTEGER*4 BUT141
INTEGER*4 MOUSE1
INTEGER*4 MOUSE2
INTEGER*4 MOUSE3
INTEGER*4 LEFTMO
INTEGER*4 MIDDLE
INTEGER*4 RIGHTM
INTEGER*4 LPENBU
INTEGER*4 BPADO
INTEGER*4 IKKEY
INTEGER*4 JKEY
INTEGER*4 KKEY
INTEGER*4 LKEY
INTEGER*4 MKKEY
INTEGER*4 NKEY
INTEGER*4 OKEY
INTEGER*4 PKEY
INTEGER*4 QKEY
INTEGER*4 RKEY
INTEGER*4 SKEY
INTEGER*4 TKEY
INTEGER*4 UKEY
INTEGER*4 VKEY
INTEGER*4 WKEY
INTEGER*4 XKEY
INTEGER*4 YKEY
INTEGER*4 ZKEY
INTEGER*4 ZERKEY
INTEGER*4 ONEKEY
INTEGER*4 TWEKEY
INTEGER*4 THREEK
INTEGER*4 FOURKE
INTEGER*4 FIVEKE
INTEGER*4 SIXKEY
INTEGER*4 SEVENK
INTEGER*4 EIGHTK
INTEGER*4 NINEK
INTEGER*4 BREAKK
INTEGER*4 SETUPK
INTEGER*4 CTRLKE
INTEGER*4 CAPSLO
INTEGER*4 RIGHTS
INTEGER*4 LEFTSH
INTEGER*4 NOSCRL
INTEGER*4 ESCKEY
INTEGER*4 TABKEY
INTEGER*4 RETKEY
INTEGER*4 SPACEK
INTEGER*4 LINEFE
INTEGER*4 BACKSP
INTEGER*4 DELKEY
INTEGER*4 SEMICO
INTEGER*4 PERIOD
INTEGER*4 COMMAK
INTEGER*4 QUOTEK
INTEGER*4 ACCENT
INTEGER*4 MINUSK
INTEGER*4 VIRGUL
INTEGER*4 BACKSL
INTEGER*4 EQUALK
INTEGER*4 LEFTBR
INTEGER*4 RIGHTB
INTEGER*4 LEFTAR
INTEGER*4 DOWNAR
INTEGER*4 RIGHTA
INTEGER*4 UPARR0
INTEGER*4 PAD0
INTEGER*4 PAD1
INTEGER*4 PAD2
INTEGER*4 PAD3
INTEGER*4 PAD4
INTEGER*4 PAD5
INTEGER*4 PAD6
INTEGER*4 PAD7
INTEGER*4 PAD8
INTEGER*4 PAD9
INTEGER*4 PADPF1
INTEGER*4 PADPF2
INTEGER*4 PADPF3
INTEGER*4 PADPF4
INTEGER*4 PADPER
INTEGER*4 PADMIN
INTEGER*4 PADCOM
INTEGER*4 PADENT

C  screen buttons
C  Hashing algorithm: SBUTx = SCRBUtx

INTEGER*4 SBUTO
INTEGER*4 SBUT1
INTEGER*4 SBUT2
INTEGER*4 SBUT3
INTEGER*4 SBUT4
INTEGER*4 SBUT5
INTEGER*4 SBUT6
INTEGER*4 SBUT7
INTEGER*4 SBUT8
INTEGER*4 SBUT9
INTEGER*4 SBUT10
C valuators

INTEGER*4 SGIRES
INTEGER*4 DIAL0
INTEGER*4 DIAL1
INTEGER*4 DIAL2
INTEGER*4 DIAL3
INTEGER*4 DIAL4
INTEGER*4 DIAL5
INTEGER*4 DIAL6
INTEGER*4 DIAL7
INTEGER*4 DIAL8
INTEGER*4 MOUSEX
INTEGER*4 MOUSEY
INTEGER*4 LPEMX
INTEGER*4 LPEMY
INTEGER*4 BPADX
INTEGER*4 BPADY
INTEGER*4 NULLX
INTEGER*4 NULLY

C timers

INTEGER*4 TIMER0
INTEGER*4 TIMER1
INTEGER*4 TIMER2
INTEGER*4 TIMER3
INTEGER*4 TIMER4
INTEGER*4 TIMER5
INTEGER*4 TIMER6
INTEGER*4 TIMER7

C misc devices

C Hash algorithm: CURSRX = CURSORX

INTEGER*4 KEYBD
INTEGER*4 CURSRX
INTEGER*4 CURSRY
INTEGER*4 VALMAR
INTEGER*4 GERROR
INTEGER*4 REDRAW
INTEGER*4 WSEND
INTEGER*4 WNREPL
INTEGER*4 WMGFCL
INTEGER*4 WMTXCL

C include file with device definitions

PARAMETER ( NULLDE = 0 )
PARAMETER ( BUTOFF = 1 )
PARAMETER ( SBTOFF = 200 )
PARAMETER ( VALOFF = 256 )
PARAMETER ( KEYOFF = 512 )
PARAMETER ( TIMOFF = 515 )

PARAMETER ( BUTCOU = 144 )
PARAMETER ( SBTCOU = 16 )
PARAMETER ( VALCOU = 14 )
PARAMETER ( TIMCOU = 8 )

C buttons

PARAMETER ( BUT0 = 1 )
PARAMETER ( BUT1 = 2 )
PARAMETER ( BUT2 = 3 )
PARAMETER ( BUT3 = 4 )
PARAMETER ( BUT4 = 5 )
PARAMETER ( BUT5 = 6 )
PARAMETER ( BUT6 = 7 )
PARAMETER ( BUT7 = 8 )
PARAMETER ( BUT8 = 9 )
PARAMETER ( BUT9 = 10 )
PARAMETER ( BUT10 = 11 )
PARAMETER ( BUT11 = 12 )
PARAMETER ( BUT12 = 13 )
PARAMETER ( BUT13 = 14 )
PARAMETER ( BUT14 = 15 )
PARAMETER ( BUT15 = 16 )
PARAMETER ( BUT16 = 17 )
PARAMETER ( BUT17 = 18 )
PARAMETER ( BUT18 = 19 )
PARAMETER ( BUT19 = 20 )
PARAMETER ( BUT20 = 21 )
PARAMETER ( BUT21 = 22 )
PARAMETER ( BUT22 = 23 )
PARAMETER ( BUT23 = 24 )
PARAMETER ( BUT24 = 25 )
PARAMETER ( BUT25 = 26 )
PARAMETER ( BUT26 = 27 )
PARAMETER ( BUT27 = 28 )
PARAMETER ( BUT28 = 29 )
PARAMETER ( BUT29 = 30 )
PARAMETER ( BUT30 = 31 )
PARAMETER ( BUT31 = 32 )
PARAMETER ( BUT32 = 33 )
PARAMETER ( BUT33 = 34 )
PARAMETER ( BUT34 = 35 )
PARAMETER ( BUT35 = 36 )
PARAMETER ( BUT36 = 37 )
PARAMETER ( BUT37 = 38 )
PARAMETER ( BUT38 = 39 )
PARAMETER ( BUT39 = 40 )
PARAMETER ( BUT40 = 41 )
PARAMETER ( BUT41 = 42 )
PARAMETER ( BUT42 = 43 )
PARAMETER ( BUT43 = 44 )
PARAMETER ( BUT44 = 45 )
PARAMETER ( BUT45 = 46 )
PARAMETER ( BUT46 = 47 )
PARAMETER ( BUT47 = 48 )
PARAMETER ( BUT48 = 49 )
PARAMETER ( BUT49 = 50 )
PARAMETER ( BUT50 = 51 )
PARAMETER ( BUT51 = 52 )
PARAMETER ( BUT52 = 53 )
PARAMETER ( BUT53 = 54 )
PARAMETER ( BUT54 = 55 )
PARAMETER ( BUT55 = 56 )
PARAMETER ( BUT56 = 57 )
PARAMETER ( BUT57 = 58 )
PARAMETER ( BUT58 = 59 )
PARAMETER ( BUT59 = 60 )
PARAMETER ( BUT60 = 61 )
PARAMETER ( BUT61 = 62 )
PARAMETER ( BUT62 = 63 )
PARAMETER ( BUT63 = 64 )
PARAMETER ( BUT64 = 65 )
PARAMETER ( BUT65 = 66 )
PARAMETER ( BUT66 = 67 )
PARAMETER ( BUT67 = 68 )
PARAMETER ( BUT68 = 69 )
PARAMETER ( BUT69 = 70 )
PARAMETER ( BUT70 = 71 )
PARAMETER ( BUT71 = 72 )
PARAMETER ( BUT72 = 73 )
PARAMETER ( BUT73 = 74 )
PARAMETER ( BUT74 = 75 )
PARAMETER ( BUT75 = 76 )
PARAMETER ( BUT76 = 77 )
PARAMETER ( BUT77 = 78 )
PARAMETER ( BUT78 = 79 )
PARAMETER ( BUT79 = 80 )
PARAMETER ( BUT80 = 81 )
PARAMETER ( BUT81 = 82 )
PARAMETER ( BUT82 = 83 )
PARAMETER ( MAXKBD = 83 )
PARAMETER ( BUT100 = 101 )
PARAMETER ( BUT101 = 102 )
PARAMETER ( BUT102 = 103 )
PARAMETER ( BUT110 = 111 )
PARAMETER ( BUT111 = 112 )
PARAMETER ( BUT112 = 113 )
PARAMETER ( BUT113 = 114 )
PARAMETER ( BUT114 = 115 )
PARAMETER ( BUT115 = 116 )
PARAMETER ( BUT116 = 117 )
PARAMETER ( BUT117 = 118 )
PARAMETER ( BUT118 = 119 )
PARAMETER ( BUT119 = 120 )
PARAMETER ( BUT120 = 121 )
PARAMETER ( BUT121 = 122 )
PARAMETER ( BUT122 = 123 )
PARAMETER ( BUT123 = 124 )
PARAMETER ( BUT124 = 125 )
PARAMETER ( BUT125 = 126 )
PARAMETER ( BUT126 = 127 )
PARAMETER ( BUT127 = 128 )
PARAMETER ( BUT128 = 129 )
PARAMETER ( BUT129 = 130 )
PARAMETER ( BUT130 = 131 )
PARAMETER ( BUT131 = 132 )
PARAMETER ( BUT132 = 133 )
PARAMETER ( BUT133 = 134 )
PARAMETER ( BUT134 = 135 )
PARAMETER ( BUT135 = 136 )
PARAMETER ( BUT136 = 137 )
PARAMETER ( BUT137 = 138 )
PARAMETER ( BUT138 = 139 )
PARAMETER ( BUT139 = 140 )
PARAMETER ( BUT140 = 141 )
PARAMETER ( BUT141 = 142 )
PARAMETER ( MOUSE1 = 101 )
PARAMETER ( MOUSE2 = 102 )
PARAMETER ( MOUSE3 = 103 )
PARAMETER ( LEFTMO = 103 )
PARAMETER ( MIDDLE = 102 )
PARAMETER ( RIGHTM = 101 )
PARAMETER ( LPENBU = 104 )
PARAMETER ( BPADO = 105 )
PARAMETER ( BPAD1 = 106 )
PARAMETER ( BPAD2 = 107 )
PARAMETER ( BPAD3 = 108 )
PARAMETER ( SWBASE = 111 )
PARAMETER ( SW0 = 111 )
PARAMETER ( SW1 = 112 )
PARAMETER ( SW2 = 113 )
PARAMETER ( SW3 = 114 )
PARAMETER ( SW4 = 115 )
PARAMETER ( SW5 = 116 )
PARAMETER ( SW6 = 117 )
PARAMETER ( SW7 = 118 )
PARAMETER ( SW8 = 119 )
PARAMETER ( SW9 = 120 )
PARAMETER ( SW10 = 121 )
PARAMETER ( SW11 = 122 )
PARAMETER ( SW12 = 123 )
PARAMETER ( SW13 = 124 )
PARAMETER ( SW14 = 125 )
PARAMETER ( SW15 = 126 )
PARAMETER ( SW16 = 127 )
PARAMETER ( SW17 = 128 )
PARAMETER ( SW18 = 129 )
PARAMETER ( SW19 = 130 )
PARAMETER ( SW20 = 131 )
PARAMETER ( SW21 = 132 )
PARAMETER ( SW22 = 133 )
PARAMETER ( SW23 = 134 )
PARAMETER ( SW24 = 135 )
PARAMETER (SW25 = 136)
PARAMETER (SW26 = 137)
PARAMETER (SW27 = 138)
PARAMETER (SW28 = 139)
PARAMETER (SW29 = 140)
PARAMETER (SW30 = 141)
PARAMETER (SW31 = 142)
PARAMETER (AKEY = 11)
PARAMETER (BKEY = 36)
PARAMETER (CKEY = 28)
PARAMETER (DKEY = 18)
PARAMETER (EKEY = 17)
PARAMETER (FKEY = 19)
PARAMETER (GKEY = 26)
PARAMETER (HKEY = 27)
PARAMETER (IKEY = 40)
PARAMETER (JKEY = 34)
PARAMETER (KKEY = 35)
PARAMETER (LKEY = 42)
PARAMETER (MKEY = 44)
PARAMETER (NKEY = 37)
PARAMETER (OKEY = 41)
PARAMETER (PKEY = 48)
PARAMETER (QKEY = 10)
PARAMETER (RKEY = 24)
PARAMETER (SKEY = 12)
PARAMETER (TKEY = 25)
PARAMETER (UKEY = 33)
PARAMETER (VKEY = 29)
PARAMETER (WKEY = 16)
PARAMETER (XKEY = 21)
PARAMETER (YKEY = 32)
PARAMETER (ZKEY = 20)
PARAMETER (ZEROKE = 46)
PARAMETER (ONEKEY = 8)
PARAMETER (TWOKEY = 14)
PARAMETER (THREEK = 15)
PARAMETER (FOURKE = 22)
PARAMETER (FIVEKE = 23)
PARAMETER (SIXKEY = 30)
PARAMETER (SEVENK = 31)
PARAMETER (EIGHTK = 38)
PARAMETER (NINEKE = 39)
PARAMETER (BREAKK = 1)
PARAMETER (SETUPK = 2)
PARAMETER (CTRLKE = 3)
PARAMETER (CAPSLO = 4)
PARAMETER (RIGHTS = 5)
PARAMETER (LEFTSH = 6)
PARAMETER (NOSCRL = 13)
PARAMETER (ESCKEY = 7)
PARAMETER (TABKEY = 9)
PARAMETER (REKEY = 51)
PARAMETER (SPACEK = 83)
PARAMETER (LINEFE = 60)
PARAMETER (BACKSP = 61)
PARAMETER (DELKEY = 62)
PARAMETER (SEMIC = 43)
PARAMETER (PERIOD = 52)
PARAMETER (COMMAK = 45)
PARAMETER (QUOTEK = 50)
PARAMETER (ACCENT = 55)
PARAMETER (MINUSK = 47)
PARAMETER (VIRGUL = 53)
PARAMETER (BACKSL = 57)
PARAMETER (EQUALK = 54)
PARAMETER (LEFTBR = 49)
PARAMETER (RIGHTB = 56)
PARAMETER (LEFTAR = 73)
PARAMETER (DOWNAR = 74)
PARAMETER (RIGHTA = 80)
PARAMETER (UPARRO = 81)
PARAMETER (PAD0 = 59)
PARAMETER (PAD1 = 56)
PARAMETER (PAD2 = 64)
PARAMETER (PAD3 = 65)
PARAMETER (PAD4 = 63)
PARAMETER (PAD5 = 69)
PARAMETER (PAD6 = 70)
PARAMETER (PAD7 = 67)
PARAMETER (PAD8 = 68)
PARAMETER (PAD9 = 75)
PARAMETER (PADPF1 = 72)
PARAMETER (PADPF2 = 71)
PARAMETER (PADPF3 = 79)
PARAMETER (PADPF4 = 78)
PARAMETER (PADPER = 66)
PARAMETER (PADMIN = 76)
PARAMETER (PADCOM = 77)
PARAMETER (PADENT = 82)
C screen buttons

PARAMETER (SBUT0 = 200)
PARAMETER (SBUT1 = 201)
PARAMETER (SBUT2 = 202)
PARAMETER (SBUT3 = 203)
PARAMETER (SBUT4 = 204)
PARAMETER (SBUT5 = 205)
PARAMETER (SBUT6 = 206)
PARAMETER (SBUT7 = 207)
PARAMETER (SBUT8 = 208)
PARAMETER (SBUT9 = 209)
PARAMETER (SBUT10 = 210)
PARAMETER (SBUT11 = 211)
PARAMETER (SBUT12 = 212)
PARAMETER (SBUT13 = 213)
PARAMETER (SBUT14 = 214)
PARAMETER (SBUT15 = 215)
PARAMETER (SBUT16 = 216)

C valuators

PARAMETER (SCIRES = 256)
PARAMETER (DIAL0 = 257)
PARAMETER (DIAL1 = 258)
PARAMETER (DIAL2 = 259)
PARAMETER (DIAL3 = 260)
PARAMETER (DIAL4 = 261)
PARAMETER (DIAL5 = 262)
PARAMETER (DIAL6 = 263)
PARAMETER (DIAL7 = 264)
PARAMETER (DIAL8 = 265)
PARAMETER (MOUSEX = 266)
PARAMETER (MOUSEY = 267)
PARAMETER (LPENX = 268)
PARAMETER (LPENY = 269)
PARAMETER (BPADX = 270)
PARAMETER (BPADY = 271)
PARAMETER (NULLX = 270)
PARAMETER (NULLY = 271)

C timers

PARAMETER (TIMERO = 515)
PARAMETER ( TIMER1 = 516 )
PARAMETER ( TIMER2 = 517 )
PARAMETER ( TIMER3 = 518 )
PARAMETER ( TIMER4 = 519 )
PARAMETER ( TIMER5 = 520 )
PARAMETER ( TIMER6 = 521 )
PARAMETER ( TIMER7 = 522 )

C misc devices

PARAMETER ( KEYBD = 513 )
PARAMETER ( CURSRX = 526 )
PARAMETER ( CURSRY = 527 )
PARAMETER ( VALMAR = 523 )
PARAMETER ( GERROR = 524 )
PARAMETER ( REDRAW = 528 )
PARAMETER ( TMSEND = 529 )
PARAMETER ( TMREPL = 530 )
PARAMETER ( TMGETL = 531 )
PARAMETER ( WXRXCL = 532 )

A. s Definitions

Pascal

const
MAXARRAY = 1023;
MAXRASTER = 4095;
MAXOBJECTS = 4095;
PATTERN_16 = 16;
PATTERN_32 = 32;
PATTERN_64 = 64;
PATTERN_16_SIZE = 16;
PATTERN_32_SIZE = 64;
PATTERN_64_SIZE = 256;


type
Byte = 0..255;
Short = integer;
UnsignedShort = 0..65535;
Angle = integer;
Sreenoord = short;
Icoord = longint;
Coord = real;
Scoord = UnsignedShort;
String = paced array [1..128] of char;
Matrix = array [0..3, 0..3] of real;

Device = UnsignedShort;
string128 = string [128];
pstring = 'Strng;

Colorindex = UnsignedShort;
RGBvalue = Byte;

Linestyle = UnsignedShort;
Pattern = array [0..15] of Byte;
Pattern16 = [0..PATTERN_16-SIZE] of UnsignedShort;
Pattern32 = [0..PATTERN_32-SIZE] of UnsignedShort;
Pattern64 = [0..PATTERN_64-SIZE] of UnsignedShort;
Cursor = array [0..15] of UnsignedShort;

Fontchar = record
  offset: Short;
  w, h: Byte;
  xoff, yoff: -128..127;
  width: Short;
end;

Object = longint;
Tag = longint;

Coord4array = array [0..MAXARRAY, 0..3] of Coord;
Coord3array = array [0..MAXARRAY, 0..2] of Coord;
Coord2array = array [0..MAXARRAY, 0..1] of Coord;
Icoord3array = array [0..MAXARRAY, 0..2] of Icoord;
Icoord2array = array [0..MAXARRAY, 0..1] of Icoord;
Scoord3array = array [0..MAXARRAY, 0..2] of Scoord;
Scoord2array = array [0..MAXARRAY, 0..1] of Scoord;
Screenarray = array [0..MAXARRAY, 0..2] of Screencoord;

Boolarray = array [0..MAXARRAY] of Boolean;
Colorarray = array [0..MAXARRAY] of Colorindex;
RGBarray = array [0..MAXARRAY] of RGBvalue;

Objarray = array [0..127] of Object;
Patcharray = array [0..127] of Fontchar;
Fontraster = array [0..MAXRASTER] of Short;
Ibuffer = array [0..MAXOBJECTS] of Object;
Queuearray = array [0..MAXARRAY] of Short;
Curvearray = array [0..3] [0..2] of Coord;
Ibuffer = array [0..MAXARRAY] of Short;
Patcharray = array [0..3] [0..3] of Coord;
Appendix B
Geometry Engine Computations

The Geometry Engine system has three parts:

- **Matrix Subsystem** - A stack of 4x4 floating point matrices transform coordinate data.
- **Clipping Subsystem** - Transformed coordinate data is clipped to a 2D or 3D window.
- **Scaling Subsystem** - Clipped 2D or 3D coordinate data is scaled to the dimensions of the output device. In 3D, the system provides orthographic or perspective projection as well.

The Matrix Subsystem does the following computation:

\[
\begin{bmatrix}
    x \\
    y \\
    z \\
    w
\end{bmatrix}
= \begin{bmatrix}
    x' \\
    y' \\
    z' \\
    w'
\end{bmatrix} M
\]

where \( M \) is the current transformation matrix and \( \begin{bmatrix}
    x' \\
    y' \\
    z' \\
    w'
\end{bmatrix} \) is the vector to be transformed. The coordinates \( \begin{bmatrix}
    x \\
    y \\
    z \\
    w
\end{bmatrix} \) are given to the Clipping Subsystem. The clipped results satisfy the following equations:

\[-w < x < w\]
\[-w < y < w\]
\[-w < z < w\]

After clipping, the Scaling Subsystem does the final mapping to screen coordinates with the following computations:

\[
X_{screen} = \frac{x}{w} \times V_{sx} + V_{cx}
\]

\[
Y_{screen} = \frac{y}{w} \times V_{sy} + V_{cy}
\]

\[
Z_{screen} = \frac{z}{w} \times V_{sz} + V_{cz}
\]

\( V_{cx} \) is the center of the viewport in the coordinate system of the output device, and \( V_{sx} \) is the horizontal distance from the center to the edge of the viewport. \( V_{cy}, V_{sy}, V_{cz}, \) and \( V_{sz} \) are similarly defined.
The IRIS user declares the viewport boundaries with the `viewport` and `setdepth` commands. The viewport centers and half-widths are computed as follows:

\[
V_{cx} = \frac{left + right}{2} \quad V_{sx} = \frac{right-left}{2}
\]

\[
V_{cy} = \frac{bottom + top}{2} \quad V_{sy} = \frac{top-bottom}{2}
\]

\[
V_{cz} = \frac{near + far}{2} \quad V_{sz} = \frac{far-near}{2}
\]
Appendix C
Transformation Matrices

Here are the matrices that are created by the transformation commands.

C.1 Translation

$$\text{Translate}(T_x, T_y, T_z) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

C.2 Scaling and Mirroring

$$\text{Scale}(S_x, S_y, S_z) = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

C.3 Rotation

$$\text{Rot}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & \sin \theta & 0 \\ 0 & -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{Rot}_y(\theta) = \begin{bmatrix} \cos \theta & 0 & -\sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{Rot}_z(\theta) = \begin{bmatrix} \cos \theta & \sin \theta & 0 & 0 \\ -\sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
C.4 Viewing Transformations

\[
\text{Polarview}(\text{dist}, \text{azim}, \text{inc}, \text{twist}) = \text{Rot}_y(-\text{azim}) \times \text{Rot}_x(-\text{inc}) \times \text{Rot}_z(-\text{twist}) \times \text{Trans}(0.0, 0.0, -\text{dist})
\]

\[
\text{Lookat}(v_x, v_y, v_z, p_x, p_y, p_z, \text{twist}) = \text{Trans}(-v_x, -v_y, -v_z) \times \text{Rot}_y(\theta) \times \text{Rot}_z(\phi) \times \text{Rot}_z(-\text{twist})
\]

where

\[
\sin(\theta) = \frac{p_z - v_z}{\sqrt{(p_z - v_z)^2 + (p_y - v_y)^2}}
\]

\[
\cos(\theta) = \frac{v_x - p_x}{\sqrt{(p_x - v_x)^2 + (p_z - v_z)^2}}
\]

\[
\sin(\phi) = \frac{v_y - p_y}{\sqrt{(p_x - v_x)^2 + (p_y - v_y)^2 + (p_z - v_z)^2}}
\]

\[
\cos(\phi) = \frac{\sqrt{(p_x - v_x)^2 + (p_y - v_y)^2}}{\sqrt{(p_x - v_x)^2 + (p_y - v_y)^2 + (p_z - v_z)^2}}
\]

C.5 Perspective Transformations

\[
\text{Perspective}(\text{fov}, \text{aspect}, \text{near}, \text{far}) = \begin{bmatrix}
\cot \left[ \frac{\text{fov}}{2} \right] & 0 & 0 & 0 \\
\text{aspect} & 0 & 0 & 0 \\
0 & \cot \left[ \frac{\text{fov}}{2} \right] & 0 & 0 \\
0 & 0 & -\frac{\text{far} + \text{near}}{\text{far} - \text{near}} & -1 \\
0 & 0 & \frac{2 \times \text{far} \times \text{near}}{\text{far} - \text{near}} & 0
\end{bmatrix}
\]
C.6 Orthographic Transformations

\[
\text{Window (left, right, bottom, top, near, far)} = \begin{bmatrix}
\frac{2 \times \text{near}}{\text{right-left}} & 0 & 0 & 0 \\
0 & \frac{2 \times \text{near}}{\text{top-bottom}} & 0 & 0 \\
\frac{\text{right+left}}{\text{top-bottom}} & \frac{\text{top+bottom}}{\text{far-near}} & -1 & 0 \\
0 & 0 & \frac{2 \times \text{far} \times \text{near}}{\text{far-near}} & 0
\end{bmatrix}
\]

\[
\text{Ortho}_{3d} (\text{left}, \text{right}, \text{bottom}, \text{top}, \text{near}, \text{far}) = \begin{bmatrix}
\frac{2}{\text{right-left}} & 0 & 0 & 0 \\
0 & \frac{2}{\text{top-bottom}} & 0 & 0 \\
0 & 0 & 0 & \frac{2}{\text{far-near}} \\
\frac{\text{right+left}}{\text{top-bottom}} & \frac{\text{top+bottom}}{\text{far-near}} & -1 & 0
\end{bmatrix}
\]

\[
\text{Ortho}_{2d} (\text{left}, \text{right}, \text{bottom}, \text{top}) = \begin{bmatrix}
\frac{2}{\text{right-left}} & 0 & 0 & 0 \\
0 & \frac{2}{\text{top-bottom}} & 0 & 0 \\
0 & 0 & -1 & 0 \\
\frac{\text{right+left}}{\text{top-bottom}} & \frac{\text{top+bottom}}{\text{top-bottom}} & 0 & 1
\end{bmatrix}
\]
Appendix D
Feedback Parser

This feedback parser simplifies the use of the Geometry Engines in feedback mode. (See Chapter 10 for a discussion of the IRIS feedback utility.) The purpose of the parser is to pass to an application program the commands and data it is interested in while discarding unwanted commands.

The parser accepts a buffer of feedback data and parses it in normal or debugging mode. In normal mode the parser passes the indicated commands and data to the application. In debugging mode the parser produces a printed trace of all commands and data not passed to the application. If no commands are requested by the application then the entire feedback buffer is parsed and printed.

The interface to the parser consists of four routines. parsefb parses a feedback buffer. Its arguments are the name of the buffer, the size of the buffer, a flag indicating whether z-buffering is on, and a flag indicating whether depth-cueing is on. parsefb returns TRUE or FALSE to indicate whether the buffer was successfully parsed.

```
parsefb(buffer, count, zflag, depthflag)
short buffer[], count;
Boolean zflag, depthflag;

(not implemented in FORTRAN)

(not implemented in Pascal)
```

setfbd debugging turns on and off the debugging trace.

```
setfbd debugging(flag)
Boolean flag;

(not implemented in FORTRAN)

(not implemented in Pascal)
```

bindfbdfunc associates an application-defined handling routine with a particular command.
bindfbbfunc(command, handlingfunc)
short command;
int (*handlingfunc)();

(not implemented in FORTRAN)

(not implemented in Pascal)

getfbbword is used by application-handling routines to get feedback words.

short getfbbword()

(not implemented in FORTRAN)

(not implemented in Pascal)

The following program illustrates the use of the feedback parser. First, a feedback buffer is created and then the buffer is parsed.

#include "gl.h"

#define BUFSIZE 400

main()
{
char zflag, depthflag;
short buffer[BUFSIZE];
short count;

zflag = getzbuffer();       /* set zflag to TRUE if z-buffering is on */
depthflag = getdcm();       /* set depthflag to TRUE if depth-cueing is on */
feedback(buffer, BUFSIZE);  /* enter feedback mode */

color(BLACK);
clear();

color(RED);
circi(200, 200, 200);

color(GREEN);
pnt2i(200, 200);
color(BLUE);
recti(0, 0, 400, 400);

count = endfeedback(buffer); /* exit feedback mode */

setfdebugging(TRUE); /* turn on debugging mode for parser */

if (parsefb(buffer, count, zflag, depthflag)) /* parse the feedback buffer */
    printf("no errors in parse\n");
else
    printf("error while parsing\n");

gexit();
}

This parsing utility can generate data for devices such as off-line plotters. A program can use bindffunc and getfword to extract primitives such as line and point drawing commands and output them to a plotter. The functions passed in the bindffunc are of the form

    handlineroutine(count, command, string)
    int count;
    int command;
    char *string;

where count is the number of data words associated with the command, command is the command encountered, and string is a string corresponding to the command.

The following program produces a UNIX plot file from a feedback buffer:

    #include "gl.h"
    #include "gl2cmds.h"
    #include "stdio.h"

    #define BUFFSIZE 100

    int plotpoint();
    int plotmove();
    int plotdraw();

    main()
    {
        char zflag, depthflag;
        short buffer[BUFFSIZE];
        short count;

        zflag = getzbuffer();
depthflag = getdcm();
feedback(buffer, BUFSIZE);

color(BLACK);
clear();

color(RED);
circi(200, 200, 200);

color(GREEN);
pnt2i(200, 200);

color(BLUE);
recti(0, 0, 400, 400);

count = endfeedback(buffer);

bindfbfunc(FBCpoint, plotpoint);
bindfbfunc(FBCmove, plotmove);
bindfbfunc(FBCdraw, plotdraw);

openpl();
if (parsefb(buffer, count, zflag, depthflag))
    fprintf(stderr,"no errors in parse\n");
else
    fprintf(stderr,"error while parsing\n");
closepl();

gexit();
}

static int lastx, lasty;

plotpoint(count, command, string)
    int count;
    int command;
    char *string;
{
    int i, x, y;

    x = getfbword();
    y = getfbword();
    point(x, y);
    for (i = 2; i < count; i++)
        getfbword();
plotmove(count, command, string)

```c
int count;
int command;
char *string;
{
  int i, x, y;

  lastx = getfbword();
  lasty = getfbword();
  for (i = 2; i < count; i++)
    getfbword();
}
```

plotdraw(count, command, string)

```c
int count;
int command;
char *string;
{
  int i, x, y;

  x = getfbword();
  y = getfbword();
  line(x, y, lastx, lasty);
  lastx = x;
  lasty = y;
  for (i = 2; i < count; i++)
    getfbword();
}
```

Here is the code listing for the feedback parser:

```c
/* "parse.c" */
#include "gl.h"
#include "uc4.h"

typedef struct {
  char *fbcstring;
  int two_d_count;
  int three_d_count;
  int (*handler)();
} Feedbacktable;
```
```c
#include "parsetab.h"

static short *buff_ptr;
static short debugging;
static short buff_count;
static short word_no;
static char parse_not_done;

parsefb(buffer, count, zbuff, depthcue)
  short  *buffer;
  short   count;
  short   zbuff, depthcue;
{
  int    (*handling_routine)();
  char   *FBCstring;
  short  in_3d;
  char   no_error = TRUE;
  short   opcode;
  short   highbyte;
  short   passthroughcount;
  int     paramcount;

  word_no = 0;
  parse_not_done = TRUE;
  buff_count = count;
  buff_ptr = buffer;
  in_3d = zbuff || (depthcue << 1);

  while (parse_not_done && no_error) {
    opcode = 0x30 & (highbyte = getfbword());
    highbyte = highbyte >> 8;
    if ((opcode < TABLE_END) && parse_not_done) {
      switch (opcode) {
        case FBCdrawmode :
          if (buff_ptr[1])
            in_3d |= 1;
          else
            in_3d &= "1");
          break;
        case FBCconfig :
          if (*buff_ptr & UC_DEPTHCUE)
            in_3d |= 2;
          else
```
in_3d &= (~2);
break;
case FBC_force_completion:
    passthrough_count = highbyte;
break;
}

handling_routine = feedbacktable[opcode].handler;
FBCstring = feedbacktable[opcode].fbcstring;
if (in_3d)
    paramcount = feedbacktable[opcode].three_d_count;
else
    paramcount = feedbacktable[opcode].two_d_count;
if (paramcount == -1)
    paramcount = passthrough_count;
(*handling_routine)(paramcount, opcode, FBCstring);
else
    no_error = FALSE;
}

if (parse_not_done)
    return FALSE;
else
    return TRUE;

static char *UNDEFINED = {"ignored"};
static parsefbback( count, opcode, fbcstring )
int         count;
int         opcode;
char       *fbcstring;
{
    int i;
    int inputword;

    if (debugging)
        printf("#%d: %s\n", word_no, (fbcstring?fbcstring:UNDEFINED));

    for (i = 0; i < count; i++) {
        inputword = getfbword();
        if (debugging)
            printf("\t#%d: 0x%x\n", word_no, inputword, inputword);
    }

    getfbword()
{  if (!(--buff_count))
      parse_not_done = FALSE;
  word_no++;
  return(*buff_ptr++);
}

setfbd debugging(flag)
{
  debugging = flag;
}

bindfbdfunc(entryno, function)
{
  int entryno;
  int (*function)();

  if ((entryno >= 0) && (entryno < TABLE_END)) {
    feedbacktable[entryno].handler = function;
    return TRUE;
  } else
    return FALSE;
}

/* "parsetab.h" */
#include "gl2cmds.h"

static parsefbdfunc();

/* fbname 2d shorts, 3d shorts, function */
static Feedbacktable feedbacktable[] = {
/*0*/ {"FBCinitfb", 0, 0, parsefbdfunc},
/*1*/ {0, 0, parsefbdfunc},
/*2*/ {"FBCmasklist", -1,-1, parsefbdfunc},
/*3*/ {"GEstoremm", 32,32, parsefbdfunc},
/*4*/ {"FBCrgbcolor", 3, 3, parsefbdfunc},
/*5*/ {"FBCrgbwrite", 3, 3, parsefbdfunc},
/*6*/ {"FBCsetthitmode", 0, 0, parsefbdfunc},
/*7*/ {"FBCclearhitmode", 0, 0, parsefbdfunc},
/*8*/ {"FBCforcecompletion", 0, 0, parsefbdfunc},
/*9*/ {"FBCbaseaddress", 1, 1, parsefbdfunc},
/*10*/ {"FBCcddcolor", 2, 2, parsefbdfunc},
/*11*/ {"GEstoreviewport", 12,12, parsefbdfunc},
/*12*/ {"GEreconfigure", 0, 0, parsefbdfunc},
<table>
<thead>
<tr>
<th>Path</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>/FBCdrawpixels</td>
<td>-1,-1</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCreadpixels</td>
<td>1,1</td>
<td>parsefback</td>
</tr>
<tr>
<td>/GEnoop</td>
<td>0,0</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCmove</td>
<td>2,4</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCdraw</td>
<td>2,4</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCpoint</td>
<td>2,4</td>
<td>parsefback</td>
</tr>
<tr>
<td>/GECurve</td>
<td>0,0</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBColor</td>
<td>1,1</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCwriten</td>
<td>1,1</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCconfig</td>
<td>2,2</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBLoadmasks</td>
<td>-1,-1</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCselectrgbcursor</td>
<td>9,9</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBClinewidth</td>
<td>1,1</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCcharposnabs</td>
<td>3,5</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCcharposnreal</td>
<td>3,5</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCdrawchars</td>
<td>-1,-1</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCselectcursor</td>
<td>7,7</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCdrawcursor</td>
<td>2,2</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCundrawcursor</td>
<td>0,0</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBClinestipple</td>
<td>2,2</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBColorstipple</td>
<td>1,1</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCsavegapeye</td>
<td>-1,-1</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCunsavegapeye</td>
<td>-1,-1</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCDepthsetup</td>
<td>6,6</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCFeedback</td>
<td>0,0</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCof</td>
<td>1,1</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCreadcharposn</td>
<td>0,0</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCCopyfont</td>
<td>3,3</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCPushname</td>
<td>1,1</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCloadname</td>
<td>1,1</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCpopname</td>
<td>0,0</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCfixcharload</td>
<td>3,3</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCfixchardraw</td>
<td>1,1</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCinitnammestack</td>
<td>0,0</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCpixelsetup</td>
<td>2,2</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCmovepoly</td>
<td>2,4</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCdrawpoly</td>
<td>2,4</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCloadram</td>
<td>-1,-1</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCclosepoly</td>
<td>0,0</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCdrawmode</td>
<td>2,2</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCsetintensity</td>
<td>1,1</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCsetbackfacing</td>
<td>1,1</td>
<td>parsefback</td>
</tr>
<tr>
<td>/BPDrawbus</td>
<td>-1,-1</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBXformpt</td>
<td>8,8</td>
<td>parsefback</td>
</tr>
<tr>
<td>/FBCBlockfill</td>
<td>4,4</td>
<td>parsefback</td>
</tr>
</tbody>
</table>
/*58*/{"FBCdumpram", -1,-1, parsefeedback},
/*59*/{"FBCzshade_scanline", 10,10, parsefeedback},
/*60*/{"BPCcommand", -1,-1, parsefeedback},
/*61*/{"FBCcopy_screen", 5,5, parsefeedback},
/*62*/{"FBCinit_scanline", 1,1, parsefeedback}
};
#define TABLE_END 63
Index

A
arc 3-10
arcf 3-10
aspect 4-5, 4-6, 8-4
asynchronous 7-1
attachcursor 6-17, 7-3
attribute 1-2, 2-5, 3-1, 3-2, 5-1, 5-4, 6-1, 6-14, 8-3, 10-2
attribute-setting 3-1

B
backbuffer 6-3
backface 1-2, 12-5
backfacing 1-2, 12-1, 12-5
baselines 5-7
basis 11-1, 11-2, 11-3, 11-4, 11-5, 11-6, 11-7, 11-8, 11-9, 11-10, 11-11, 11-12, 11-13, 11-14, 11-15, 11-18, 11-19, 11-20, 11-21
basisid 11-6
bbox2 8-4
bell 7-1, 7-9
Bezier 8-11, 11-3, 11-4, 11-9, 11-10, 11-11, 11-12, 11-13, 11-14, 11-15, 11-17, 11-18, 11-19, 11-20
beziermatrix 11-8, 11-12, 11-13, 11-17
bicubic 11-15
bit-mapped 1-1
bitplane 1-1, 1-2, 2-2, 3-13, 3-14, 6-1, 6-2, 6-5, 6-7, 6-9, 6-10, 6-11, 6-12, 6-14, 8-10
blink 6-8, 6-9
blkqread 7-6, 7-7
bounding box 5-6, 5-7, 8-4
B-spline 11-5, 11-9, 11-11, 11-12, 11-13, 11-15, 11-17, 11-18, 11-21
bsplinematrix 11-8, 11-12, 11-17
buffer 2-4, 6-1, 6-2, 6-3, 6-4, 6-5, 6-9, 6-10, 6-11, 6-16, 6-17, 9-2, 9-3, 9-4, 9-5, 10-1, 10-2, 10-3, 10-4, 10-5, 10-6, 10-8
button 2-4, 7-1, 7-2, 7-3, 7-4, 7-5, 7-9, 9-5, 9-6, 9-7

C
callfunc 8-11
callobj 8-3
Cardinal spline 11-3, 11-4, 11-5, 11-9, 11-11, 11-12, 11-13, 11-15, 11-17, 11-18, 11-21
charstr 3-1, 3-11, 5-7, 9-2
chunksize 8-9, 8-10
circ 3-9
circf 3-9
circle 1-2, 3-1, 3-9, 4-1, 5-1, 5-2, 6-9, 6-10, 10-2
clear 1-4, 3-1, 3-2, 3-3
clearhitcode 9-9
click 7-1, 7-8
clip 1-2, 10-1, 10-2
clipped 1-1, 3-12, 3-13, 4-8, 10-2, 10-4, 10-6
dippers 10-1, 10-2, 10-6
dipping 1-1, 3-1, 3-12, 4-5, 4-6, 4-7, 4-8, 4-9, 10-1, 10-2, 10-5
dkoff 7-8
dkoff 7-8
dkon 7-8
closeobj 8-1, 8-5, 8-7, 8-8, 8-9
cmov 3-1, 3-11
color 6-9
color-inverted 6-7
compactify 8-9
compaction 8-9
compilers 9-7, 11-9, 11-19
concave polygons 3-7
converters 6-1
crv 11-7, 11-11
crvm 11-11, 11-12, 11-13
culling 8-4
curorigin 6-14
current character position 3-2, 3-11, 3-12, 3-13, 3-14
current color 3-2, 3-9, 3-10, 6-7, 6-8, 6-9, 13-2
current graphics position 3-1, 3-2, 3-3, 3-4, 3-5, 3-6, 3-7, 3-8, 3-9, 3-10
cursoff 6-16
cursor 6-16
cursor 2-2, 2-4, 3-3, 3-12, 6-11, 6-13, 6-14, 6-15, 6-16, 6-17, 7-3, 7-7, 8-4,
9-1, 9-2, 9-4, 9-5
curvature 11-1, 11-15
curvebasis 11-6
curvet 11-13, 11-14
curveprecision 11-6, 11-9, 11-11
curves 1-1, 1-2, 3-1, 5-1, 5-2, 10-2, 11-1, 11-3, 11-4, 11-14
cycle 6-7
cyclemap 6-7

d
 dbtext 7-9
defbasis 11-6, 11-15
defcursor 6-13
deflinestyle 5-1
defpattern 5-5, 5-6
defrasterfont 5-6, 5-7
delay 11-11, 11-21
delete 8-7
delobj 8-2
deltag 8-6
depthcue 13-3
depth-cue 1-2, 10-5
depth-cueing 1-1
device 1-1, 1-2, 6-13, 6-17, 7-1, 7-2, 7-3, 7-4, 7-5, 7-6, 7-7, 7-8, 9-2, 10-1
dial 7-1, 7-2, 7-3, 7-9, 7-10
digital-to-analog 6-1
double buffer mode 6-1, 6-2, 6-3, 6-4, 6-5, 6-7, 6-9, 6-14, 12-1
doublebuffer 6-2
draw 1-2, 3-1, 3-3, 3-4, 3-9, 3-10, 5-2, 8-3, 8-4, 9-1, 9-2, 9-5, 9-6, 9-7, 10-1, 10-2, 11-1, 11-3, 11-9, 11-10, 11-11, 11-15, 11-21

e
 edge 3-13, 5-5
editobj 8-4, 8-5
endfeedback 10-3
endpick 9-4
endselect 9-10
endtag 8-5, 8-6, 8-7
event queue 4-2, 7-3, 7-4, 7-5, 7-6

eye-space 4-1

F
feedback 1-2, 1-3, 10-1, 10-2, 10-3, 10-4, 10-5, 10-6, 10-8
finish 8-10, 8-11
font 2-2, 2-5, 3-11, 3-12, 5-1, 5-6, 5-7, 5-8
foreground 7, 8 (Window Manager Section)
frontbuffer 6-4
frustum 4-6

G
fudge 7 (Window Manager Section)
gbegin 2-1
gconfig 6-1, 6-7
Geometry Accelerator 10-2
Geometry Engine 1-1, 10-1, 10-2
Geometry Pipeline 1-1, 10-1, 10-2, 10-3, 10-5, 10-6, 11-11, 12-2
genobj 8-2
gentag 8-6
getbuffer 6-4
getbutton 7-4
getcmmode 6-6
getcolor 6-10
getcpos 3-2
getcursor 6-14
getdcm 13-4
getdepth 12-1
getdisplaymode 6-4
getfont 5-7
getgpos 3-1
getheight 5-8
gethitcode 9-9
getlsbackup 5-4
getlsrepeat 5-5
getlstyle 5-4
getlwidth 5-5
getmap 6-7
getmatrix 4-2, 4-11
getmcolor 6-8
getmem 8-9, 8-10

IRIS User's Guide
getmonitor 2-5
getopenobj 8-9
getorigin 9  (Window Manager Section)
getpattern 5-6
getplanes 6-5
getport 7, 8, 9, 13 (Window Manager Section)
getresetls 5-4
getscrmask 4-8
getsize 8 (Window Manager Section)
getvp 14-2
getvaluator 7-3
getviewport 4-8
getwritemask 6-12
getzbuffer 12-3
gexit 2-4, 2-5
gflush 2-4
ginit 2-1, 6-9, 8-10, 11-10, 11-20
glyph 6-13, 6-14, 6-15, 7-3, 9-2, 9-4
Gouraud-shading 1-2, 13-1
graphical object 4-1, 4-2, 8-1
gross clipping 3-12
greset 2-4
gRGBcolor 6-11
gRGBcursor 6-15
gRGBmask 6-12
gsync 6-5

I
imakebackground 8  (Window Manager Section)
initnames 9-3
input/output 1-2, 7-1, 7-8
installation 9-7, 11-9, 11-11, 11-19, 11-21
intensities 1-2, 6-5, 6-8
intensity 6-1, 6-10
IRIS 1-1, 1-2, 1-4, 2-1, 2-2, 2-4, 2-5, 3-1, 4-8, 5-1, 5-3, 6-1, 6-4, 7-1, 7-7, 8-1, 8-10, 9-1, 9-2, 9-4, 9-5, 9-6, 10-1, 10-2, 10-3, 10-5, 11-1, 11-2, 11-15
isobj 8-2
istag 8-6

K
keepaspect 5  (Window Manager Section)
keyboard 1-1, 2-4, 7-1, 7-8, 7-9

L
lampoff 7-8
lampon 7-8
linestyle 2-5, 3-4, 3-6, 3-7, 3-9, 3-10, 5-1, 5-2, 5-3, 5-4, 5-5, 10-1, 11-17
linewidth 2-5, 3-4, 3-6, 3-7, 3-9, 3-10, 5-3, 5-5, 11-17
loadmatrix 4-2, 4-10, 10-2
loadname 9-3
lookat 4-2, 4-4, 4-5
lsbackup 5-2
lsrepeat 5-3

M
makeobj 8-1, 8-5, 8-10
maketag 8-5
mapcolor 6-8
mapping 1-2, 4-1, 4-5, 5-8, 6-1
maps 2-4, 3-13, 4-9, 6-2, 6-6, 6-7, 9-1, 10-4, 10-6
mapw 9-1
marker 10-3, 10-4
masks 6-12
matrix 1-1, 4-2, 4-9, 4-10, 4-11, 9-5, 10-1, 10-2, 10-6, 10-7, 10-8, 11-2, 11-3, 11-4, 11-5, 11-6, 11-7, 11-8, 11-9, 11-10, 11-12, 11-13, 11-14, 11-16, 11-17, 11-18, 11-20
maxsize 5 (Window Manager Section)
minsize 5 (Window Manager Section)
monitor 1-1, 2-2, 2-5, 5-1
mouse 7-1, 7-2, 7-3, 9-5, 9-6, 9-7
move 1-4, 3-1, 3-3, 3-4, 7-5, 8-5, 10-1, 10-2, 11-8, 11-13
multimap 6-6, 6-7, 6-8, 6-9
multimatrix 4-2, 4-10
N
network 7-7, 8-10
newtag 8-5
nextmap 6-7
noise 7-5
noport 7 (Window Manager Section)
NTSC 2-5

O
objdelete 8-7, 8-8
objects 0-1, 1-1, 1-2, 2-1, 4-1, 4-2, 4-3, 4-5, 8-1, 8-2, 8-3, 8-4, 8-5, 8-9, 9-1, 9-2, 9-3, 9-5
objinsert 8-7, 8-8
objreplace 8-7, 8-8
onemap 6-6, 6-7, 6-9
ortho 4-2, 4-5, 4-6

P
pagecolor 14-3
pagewritemask 14-3
passed-through 10-3
passthrough 10-2, 10-3, 10-4
patch 1-5, 11-1, 11-14, 11-15, 11-16, 11-17, 11-18, 11-19, 11-20, 11-21
patcharray 11-17
patchbasis 11-15
patchcurves 11-16
patchprecision 11-16
pclos 3-8, 3-9
pdrw 3-7, 3-8
pdrw 3-8
perspective 3-1, 4-2, 4-5, 4-6
pick 9-1, 9-2, 9-5, 9-8
picking 3-1, 3-2, 10-1, 10-4, 10-5, 10-6
pickcolor 4-2, 4-4
polf 3-7
polling 7-1, 7-7
poly 3-7
polygons 1-1, 1-2, 3-1, 3-7, 4-1, 5-1, 5-5, 10-2
popattributes 2-6, 8-3
popmatrix 4-2, 4-3, 4-11
popname 9-3
popviewport 4-9
prefposition 6 (Window Manager Section)
prefsize 6 (Window Manager Section)
pre-multiply 4-10
processor 1-1, 2-2, 10-1, 10-2
projection 2-4, 4-2, 4-4, 4-5, 4-6, 4-9, 4-10, 9-1, 9-5, 9-6, 10-2, 10-4, 10-5
pruning 8-4
pushattributes 2-5, 8-3
pushmatrix 4-2, 4-3, 4-10, 4-11
pushname 9-3
pushviewport 4-8

Q
qdevice 7-4
qenter 7-6
qread 7-6
qreset 7-7
qtest 7-6
qtype 7-6

R
raster 1-1, 5-6, 5-7, 5-8, 10-1
rdr 3-5
readpixels 3-13
readRGB 3-14
rect 3-5, 3-6
rectcopy 3-6
rectf 3-5, 3-6
reset 5-2, 5-3
reshapeviewport 9 (Window Manager Section)
RGB 2-5, 3-13, 6-1, 6-2, 6-3, 6-4, 6-5, 6-6, 6-8, 6-10, 6-11, 6-12, 6-14, 6-15, 6-16
RGBcolor 6-10
RGBcursor 6-15
RGBmode 6-2
RGBvalue 3-13, 3-14, 6-9, 6-9, 6-15
RGBwritemask 6-12
index

ringbell 7-9
rmv 3-5
rotate 4-1, 4-2, 4-4, 5-2, 5-4
rpdr 3-8, 3-9
rpmv 3-8

s
scale 1-2, 4-1, 4-2, 4-3, 4-4, 10-1
scalers 10-1, 10-2, 10-6
scaling 1-1, 3-1, 4-3, 10-1
screenmask 3-12, 3-13, 4-8, 4-9
screenspace 9 (Window Manager Section)
scremask 4-2, 4-8
select 1-2, 9-1
setbell 7-9
setcursor 6-13
setdepth 12-1
setdbltlits 7-10
setlinestyle 5-1, 5-3
setmap 6-6
setmonitor 2-5
setpattern 5-6
setshade 13-2
setvaluator 7-2
shaderange 13-4
shading 1-1, 1-2, 3-1
singlebuffer 6-2
spclos 13-2
sphere 8-1
splf 13-1
spline 11-3, 11-4, 11-5, 11-11, 11-13, 11-15, 11-21
stepunit 6, 7 (Window Manager Section)
structures 1-2
strwidth 5-8
styles 9-7, 11-9, 11-19
surfaces 1-2, 3-1, 11-1, 11-3, 11-14, 11-15
swapbuffers 6-3
swapinterval 6-3
switch 7-10, 9-6
switches 7-1, 7-10

t
tag 8-5, 8-6, 8-7, 8-8
terminal 2-4, 7-1, 7-7, 8-10
termination 6-7
text 1-2, 1-4, 1-5, 3-1, 3-11, 3-12, 5-7, 5-8, 7-1, 7-9
textcolor 14-2
textport 1-2, 3-5, 3-12
textinit 14-2, 14-3
texture 3-2, 3-6, 3-7, 3-9, 3-10, 5-1, 5-5, 10-1
textwritemask 14-2
tie 7-5
timed 6-7
tpoff 14-1
tpon 14-1
transformation 1-2, 2-4, 3-1, 4-1, 4-2, 4-4, 4-5, 4-9, 4-10, 4-11, 9-5, 9-6, 10-2, 10-4, 10-6, 10-7, 11-7, 11-8, 11-14
translate 4-1, 4-2, 4-3, 4-4

u
unqdevice 7-4
usegments 11-16, 11-18, 11-20
ushort 5-5

v
valuators 6-14, 7-1, 7-2, 7-3, 7-5
viewing 4-1, 4-2, 4-4, 4-5, 4-6, 4-9, 4-10, 9-1, 10-2
viewport 3-2, 3-12, 3-13, 3-14, 4-2, 4-5, 4-6, 4-7, 4-8, 4-9, 5-7, 8-4, 9-1, 9-5, 10-2

w
window 4-2, 4-5, 4-6
wireframe 11-15, 11-16
workstation 2-4, 8-10
writemask 2-2, 2-5, 3-2, 3-4, 3-6, 3-7, 3-9, 3-10, 6-11, 6-12, 6-13, 6-14, 6-15, 11-17
writepixels 3-13
writeRGB 3-13

version 2.1
iris user's guide
X
xfpt 10-2, 10-6
xmaxscreen 1-4, 2-4, 3-13, 9-6

Y
yamaxscreen 2-4, 9-6

Z
z-buffer 1-2, 6-1, 10-5, 12-1
z-buffering 6-2, 12-1
zbuffer 12-2
zclear 12-2
Multiple Exposure: The IRIS Window Manager

1. Introduction

Multiple Exposure is the window manager for the IRIS workstation. It provides the user with multiple concurrent windows, or ports, for text and graphics output. These windows can be of various sizes and locations, and can overlap each other. The availability of multiple windows, in effect, creates multiple graphics systems on a single IRIS workstation.

Normally, an IRIS workstation provides a single rectangular area of the screen, called the textport, for interacting with system utilities and creating graphics programs. With the IRIS window manager, however, you can create multiple textports, each running its own UNIX shell. You use the mouse to specify the size and location of each new textport and to attach input from the keyboard to a textport. Whenever a graphics program is called from a textport, it draws to an area of the screen called a graphics port. The size and location of a graphics port either is specified in the program or is specified by the user with the mouse when the program is executed. Graphics programs in several textports can be run simultaneously, each drawing to a separate graphics port. Textports and graphics ports may be moved and reshaped after they have been created, and they may overlap each other.

2. The Default User Interface to the Window Manager

This section describes the default user interface to the window manager: how to start and stop the window manager, and how to communicate with the window manager to move, reshape, create, and delete windows. Section 6 (Customizing the User Interface) describes how to configure more complex interfaces.

Starting the window manager

When an IRIS is booted, a single textport appears on the screen. The size and location of this textport can be changed by a graphics program, using the textport commands discussed in Chapter 14 of the IRIS Programming Guide. The window manager program is called mex (for Multiple EXposure) and is located in /usr/bin. You start the window manager by typing "mex<return>". The existing textport is put into a window called the console, which is the same size and shape as the original textport. The console can be used to create, compile, and execute graphics programs, and to interact with any other system utilities. You can change the size and location of the console, but you cannot delete it. After mex has been called, you can create additional textports and run graphics programs which create graphics ports.
Communicating with the window manager

The right mouse button is the primary means of communication between the user and the window manager. The other buttons are ignored by the window manager in the default state. Pushing down on the right mouse button suggests an action. The window manager presents a menu or uses the shape of the cursor glyph to indicate the action. Letting up on the mouse button confirms the action.

When the right button initially is pressed, one of two menus will appear. If the cursor is within a window, this menu appears:

```
| kill  |
| push |
| pop  |
| reshape |
| move |
| attach |
```

If the cursor is located outside all windows (that is, in the background), this menu appears:

```
| exit |
| new shell |
| attach |
```

A third menu appears to confirm the selection of "kill" or "exit":

```
| Nah...forget it |
| Do it. I'm sure |
```

Creating new textports and selecting textports for input

You can create additional textports by pressing the right mouse button while the cursor is over the screen background (not in a window). The background menu described above will then appear on the screen. If "new shell" is selected, a red bracket will appear and will follow the movement of the cursor. Pressing the right button again sets the location of one corner of the new window. Releasing the button sets the opposite corner of the window. The new textport will run the UNIX shell located in `/bin/csh`. In the standard configuration, up to ten textports can be created.

Since the keyboard and other input devices are shared among all the windows on the screen, it is necessary to specify the window for which input is intended. To do this, move the cursor into the chosen window and press the right mouse
button. When the window menu appears, select "attach". The chosen window is then activated for interaction with the user. Characters typed on the keyboard will be echoed in that textport and interpreted by that textport's shell.

You can also send input to the window manager by positioning the cursor over the background and selecting "attach". In this state, all the mouse button events (not just the right button) and any other non-textual input are put in the window manager's event queue. Input from the keyboard will be sent to the most recently selected textport.

**Moving and resizing windows**

To move windows, position the cursor in the window and press the right button. When the window menu appears, select "move" and release the button. The cursor will appear as four small red arrows. A copy of the window boundary will appear in red. The red outline will move along with the movement of the cursor. However, note that the outline always remains entirely within the boundaries of the screen. When the right mouse button is pressed and released again, the window is redrawn in the location identified by the red outline.

To change the shape of a textport, select "reshape". A second press of the right button indicates the location of one corner of the window, and a release indicates the location of the opposite corner.

**Popping and pushing textports**

The windows on the IRIS screen may overlap each other (like papers on a desktop). Two of the window menu items, "push" and "pop", allow you to arrange the order in which the windows are plastered on the screen. If you select "pop" while the cursor is located inside a particular window, that window is drawn on top of all the windows it may overlap. If you select "push", the window is placed beneath all of the windows it overlaps (i.e., the other windows are redrawn on top of the selected window).

**Running graphics programs**

You can run graphics programs by attaching input to a textport (as described above) and typing the name of the program into the textport. The output of a graphics program will appear on the screen in a graphics port. The size and location of the graphics port is determined by routines (described in Section 3) contained at the beginning of the program. However, not all of that information needs to be specified in the program. The window manager will prompt the user for whatever is missing. For example, if an application needs a square graphics port but the size and location does not matter, only the shape of the graphics port is specified in the program. When the program is called, the cursor prompts the user for the size and location of the graphics port. The square shape is maintained.

Graphics ports can be moved, reshaped, pushed, and popped (just like textports) using the right mouse button and the window menu. If a graphics program requires interaction with the user, you can attach input to the
program's graphics port by selecting the "attach" menu item. All input from available devices (e.g., keyboard and mouse) will then be interpreted by the graphics program. In the standard configuration, the right mouse button is reserved for interaction with the window manager and is not available to graphics programs.

Removing textports and graphics ports from the screen

To remove a window from the screen, move the cursor into the chosen window and press the right mouse button. Then select "kill" and release the button. A confirmation menu will appear. If you are sure you want to remove the window, select "Do it. I'm sure". When a window is removed, any active processes associated with the window will be killed. Also, if you exit from a shell its textport will be removed and if you exit from a graphics program its graphics port will be removed. Note that the console textport cannot be removed.

Leaving the window manager

To leave the window manager, position the cursor over the screen background and press the right mouse button. When the background menu appears, select "exit". When the confirmation menu appears, select "Do it. I'm sure".

3. Programming the Window Manager

Since the contents of a graphics port can only be created by a graphics program, the graphics port itself is created at the beginning of a program. Graphics programs have varying screen requirements. Some may need space with a specific aspect ratio (shape); some may have a minimum or maximum size; some may have their size or location fixed; and some don't need any screen space at all (for example, if the program only sets the color map).

The following commands are contained in the Graphics Library. They are divided into three categories: 1) commands for specifying graphics port characteristics; 2) commands for creating graphics ports; and 3) various graphics port utility commands.

Specifying graphics port characteristics

These commands are given before the graphics port is created. None of them are required; any or all of them can be given, in any order. If a characteristic is not specified, the window manager will interact with the user for the necessary information when the graphics port is created. This interaction follows the same conventions as the user interface described in the previous section. For example, if no characteristics are specified, the cursor appears as a red bracket when the graphics port is created and the user presses the right mouse button to locate one corner of the graphics port and releases the button to locate the opposite corner.
\texttt{minsize(x, y)}
\texttt{long x, y;}

\texttt{subroutine minsz(x, y)}
\texttt{integer*4 x, y}

\texttt{procedure minsize(x, y: long);}

\texttt{minsize} specifies a minimum size for the graphics port. The graphics port cannot be reshaped to be smaller than this minimum size. The default minimum size is 40 pixels wide and 30 pixels high.

\texttt{maxsize(x, y)}
\texttt{long x, y;}

\texttt{subroutine maxsz(x, y)}
\texttt{integer*4 x, y}

\texttt{procedure maxsize(x, y: long);}

\texttt{maxsize} specifies a maximum size for the graphics port. The default maximum size is 1024 pixels wide and 768 pixels high.

\texttt{keepaspect(x, y)}
\texttt{long x, y;}

\texttt{subroutine keepas(x, y)}
\texttt{integer*4 x, y}

\texttt{procedure keepaspect(x, y: long);}

\texttt{keepaspect} specifies the aspect ratio (height-to-width ratio) of the graphics port. If the graphics port is resized, its size will change but its shape will remain the same. \texttt{keepaspect(1,1)} will ensure that the graphics port is always square.


\begin{verbatim}
prefsize(x, y)
  long x, y;

subroutine prefsi(x, y)
  integer*4 x, y

  procedure prefsi(x, y: long);

\end{verbatim}

\textit{prefsize} specifies that the size of the graphics port should be \textit{x} pixels by \textit{y} pixels. The window manager will not allow the graphics port to be resized.

\begin{verbatim}
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: long);

\end{verbatim}

\textit{prefposition} specifies the preferred location and size of the graphics port. The window manager will not allow the graphics port to be relocated.

\begin{verbatim}
stepunit(xunit, yunit)
  long xunit, yunit;

subroutine stepun(xunit, yunit)
  integer*4 xunit, yunit

  procedure stepunit(xunit, yunit: long);

\end{verbatim}

\textit{stepunit} makes the graphics port change size in steps of \textit{xunit} and \textit{yunit}. For example, an \textit{xunit} of five would make it convenient to draw five equally sized columns in the graphics port, since the width of the graphics port would always be a multiple of five.
fudge(xfudge, yfudge)
long xfudge, yfudge;

subroutine fudge(xfudge, yfudge)
integer*4 xfudge, yfudge

procedure fudge(xfudge, yfudge; long);

fudge specifies that these fudge values should be added to the port dimensions as it is resized. This is useful for adjusting the size of a graphics port so that it is convenient for the program to draw a border or add a heading. fudge is often used in conjunction with stepunit. For example, specifying an xunit of five and an xfudge of two would make it convenient to create five equal columns with a small border around them.

noport()

subroutine noport

procedure noport;

noport specifies that the application does not need any screen space at all. This is useful for programs which only read or write the color map. A call to getport is still needed to do a graphics initialization (see the following section).

foreground()

subroutine foregr

procedure foreground;

foreground specifies that the graphics program will not be run in background when getport is called.
Creating a graphics port

```c
getport(name)
char name[];

subroutine getpor(name)
character*(*) name

procedure getport(name: pstring);
```

getport is called after all the graphics port characteristics have been specified. getport does a graphics initialization and then has the window manager select a region on the screen. It also puts the program in background, unless the foreground command (see previous section) has been called. The default viewport is set to the size of the graphics port.

```c
imakebackground()

subroutine imakeb

procedure imakebackground;
```

imakebackground is called instead of the characteristics given above and getport to set up a process which draws the background of the window manager’s screen. The process should draw the background and then read the input queue. Every time there is a REDRAW event in the input queue, the process redraws the background.

Graphics port utilities

```c
getsize(x, y)
long *x, *y;

subroutine getsiz(x, y)
integer*4 x, y

procedure getsize(x, y: long);
```

getsize returns the size of the graphics port. It should only be called after getport.
getorigin(x, y)
long *x, *y;

subroutine getori(x, y)
integer*4 x, y

procedure getorigin(x, y: long);

getorigin returns the position of the lower left corner of the graphics port. It should only be called after getport.

screenspace()

subroutine screen

procedure screenspace;

screenspace puts the program in screen space: graphics positions are expressed in absolute screen coordinates. (0, 0) is the lower left corner of the screen while (1023, 767) is the upper right corner of the screen. This allows you to read pixels and locations outside of the graphics port.

reshapeviewport()

subroutine reshap

procedure reshapeviewport;

reshapeviewport sets the viewport to current dimensions of the graphics port.
4. Programming Examples

The first example uses single buffer display mode and the second example uses double buffer display mode.

```c
/*
  * zoing - make a spiral out of circles
  */
#include "device.h"
#include "gl.h"

main()
{
    short val;

    keepaspect(1,1); /* the graphics port can be any location and
                     size, as long as it's square */
    getport("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(0.0,0.0,0.1.0);
        popmatrix();
    }
    popmatrix();
```

}  
    popmatrix();  
}

/*
 *    A double buffered window manager program
 *    Draws a cube which is rotated by movements of the mouse.
 *    */
#include "gl.h"
#include "device.h"

main()
{
    int x, y;    /* current rotation of object*/
    int active;    /* TRUE if window is attached*/
    short dev, val;

    keepaspect(3,2);
    getport("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);

    x = 0;  y = 0;
    active = 0;
    while(TRUE) {
        if(active) {
            frontbuffer(FALSE); /* draw scene in back buffer */
            drawcube(x, y);
            swapbuffers();
        } else {
            frontbuffer(TRUE); /* draw scene in both buffers */
            drawcube(x, y);
            while (!qtest())
                swapbuffers(); /* swap buffers if not active */
        }
        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();
}

5. Programming Considerations

1) All the windows on the screen share the same color map, texture patterns, raster fonts, and cursor glyphs. Groups of graphics programs that will be run at the same time should be designed to cooperate in their use the facilities.

2) Raster fonts are not scaled. Even though the size of a graphics port (and most of the graphics drawn inside of it) can vary, the raster font text will remain the same size.

3) Each graphics program has an event queue. A REDRAW token will appear in the owner's queue when a window is moved or reshaped, when part of a window that was obscured is revealed, or when the display mode is changed. Programs can also queue the pseudo device INPUTCHANGE. When an INPUTCHANGE event is read from the queue, a value of 0 means that the input focus has been moved to the window. A value of 0 means that input has been removed from the window. Another pseudo device, MODECHANGE, indicates that the display mode has changed from single buffer mode to double buffer mode or vice versa. When this happens, programs should do a getplanes to see how many bitplanes are available in the new display mode and perhaps change the color map to a different set of colors.

4) The window manager works only in single or double buffer mode (not in RGB mode). The window manager uses two bitplanes in single buffer mode and four bitplanes in double buffer mode. Each double buffered program is blocked until all other double buffered programs are ready to swap. Then all programs running in double buffer mode swap at the same time.

5) The getport command can replace ginit. ginit or gbegin request graphics ports with the default parameters.

6) The textport commands (tpon, tpoff, textport, gettp, textcolor, textwritемask, pagecolor, pagewritемask, textinit) do not work with the window manager.
6. Customizing the User Interface

The default window manager user interface reserves the right mouse button for its own use. Thus user applications can use only two of the mouse buttons. The default user interface was chosen primarily because of its simplicity – all window manager commands are accessed via one button, and that button always invokes a window manager command.

Many users prefer a more versatile interface. The window manager can be customized in many ways. This section describes the customization mechanism.

The .mexrc configuration file

When the window manager starts up, it looks for a file named .mexrc in your home directory. If none is found, you get the default user interface. The default user interface is defined by the default configuration file at the end of this section. There are five types of entries in a configuration file: reservebut, bindfunc, bindindex, bindcolor, and mapcolor.

Reserving buttons

reservebut reserves a button for the use of the window manager. The default user interface file contains the command:

    reservebut 101

to reserve the right mouse button for the window manager. When a button is reserved for the window manager, input events from that button are sent to the window manager process. If a textport is the active window, all non-keyboard events go to the window manager. If a graphics port is active, only the reserved buttons go to the window manager. If the window manager is active, all button events go to it.

Binding functions to buttons

bindfunc binds a window manager function to a button. Available functions include: menu, hogmode, hogwhiledown, kill, move, push, pop, attach, popattach, and movegrow. Their actions are described below:

menu: The window manager popup menu is displayed when its button is pressed. If the cursor is over a window, the large menu with kill, push, pop, reshape, move, attach is displayed; otherwise, the small menu with exit, new shell, and attach is shown.

hogmode: When the bound button is pressed (assuming it is reserved by reservebut) the window manager becomes the active process. All button clicks go to it, and continue to do so until another window is attached.

hogwhiledown: While the button bound to this command is down, all button events go to the window manager. When it is released, the window that was previously attached again gets all the keystrokes unless an attach command occurred while the button was down.
kill, move, push, pop, attach: The window under the cursor is killed, moved, pushed, popped, or attached as if the corresponding commands in the main menu were used.

popattach: This does the equivalent of a pop and an attach to the window under the cursor.

movegrow: The window under the cursor is moved or resized, depending on whether the cursor is near the center or a corner of the window. If it is near the center, the window outline is moved rigidly, tracking the cursor, as long as the button is down. When the button is released, the window is moved to the new position. When the button is pressed near a corner, only that corner is moved. All movement and reshaping are constrained by the size and aspect ratio specifications for the window.

Binding color indices to window features

bindindex assigns color indices to various parts of the window. The border is two pixels wide: the inner pixels form an inner border, and the outer pixels an outer border. The border can be highlighted or not. In addition, some windows have a title, drawn in the inner border color. The text in the title has an interior and exterior as well, which can be highlighted or not. All these regions of a window can be assigned color indices. The names of the regions are: inborder, outborder, hiinborder, hioutborder, titletextin, titletextout, hititletextin, and hititletextout. The command

bindindex hititletextin 7

in the configuration file binds the colorindex 7 to the highlighted interior of the title text.

Binding colors to window manager features

bindcolor binds RGB colors to the reserved bitplanes of the window manager – the popup menu text, popup menu background, and cursor. The commands

bindcolor menu 255 0 0
bindcolor menuback 0 255 0
bindcolor cursor 0 0 255

will cause the menu text, menu background, and cursor to be pure red, green, and blue, respectively.

Setting color map entries

mapcolor changes a color map entry to a specified RGB value. The commands

mapcolor 1 255 0 0
mapcolor 2 0 255 0
mapcolor 3 0 0 255

set the color map indices 1, 2, and 3 to be pure red, green, and blue, respectively.
An example of a different interface

The following configuration file:

```
reservebut 78
bindfunc hogmode 78
reservebut 13
bindfunc hoghiledown 13
bindfunc popattach 103
bindfunc movegrow 102
bindfunc menu 101
```

has a more interesting user interface than the default. Two buttons are reserved to it – the no-scroll key (button 13) at the lower left corner of the keyboard, and the PF4 key (button 78) at the upper right. When the PF4 key is pressed, all button events go to the window manager, including all the mouse buttons. The right mouse button brings up the menu, the center button invokes the `movegrow` command, and the left button the `popattach` command. All keystrokes go to the window manager until an `attach` command is issued (via the main popup menu).

While the no-scroll key is held down, the mouse events go to the window manager. When it is released, keystrokes revert to the window to which they were previously directed. Thus, to issue a single window manager command, the no-scroll button is easiest to use, and to issue a sequence of commands, the PF4 key is used.

All the default colors and color indices are used with this configuration file.

The default configuration file

```
reservebut 101
bindfunc menu 101
bindindex inborder 0
bindindex outborder 7
bindindex hiinborder 7
bindindex hioutborder 1
bindindex titletextin 4
bindindex titletextout 3
bindindex hititletextin 3
bindindex hititletextout 4
bindcolor menu 255 255 0
bindcolor menuback 0 0 255
bindcolor cursor 255 0 0
```
PROGRAMMING EXAMPLES

Example 1: Getting Started
Example 2: Common Drawing Commands
Example 3: Object Coordinate Systems
Example 4: Double Buffer Display Mode
Example 5: Input/Output Device
Example 6: Interactive Drawing
Example 7: Popup Menus
Example 8: Modeling Transformations
Example 9: Writemasks and Color Maps
Example 10: A Color Editor
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:

```c
/* "first.c" */
#include "gl.h"

main()
{
    glinit();
    cursoroff();
    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:

```c
#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 `glinit()` (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. `glinit()` 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 `glinit()` 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. `glinit()` 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

```cpp
    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:

```cpp
    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:

```c
/* "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":

```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);
arcf(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, 890, 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, 388);
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*i, 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":

ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
only half of it shows
CALL ARCFI(150.300.50.0.1800)
the second scoop is cherry
CALL COLOR(RED)
CALL CIRCFI(150.0.400.0.50.0)
CALL COLOR(BLACK)
outline the cone in black
CALL POLY2I(3,CONE)

next draw a few filled and unfilled arcs in the upper
left corner of the screen
CALL ARCFI(100.0.650.0.40.0.450.0.2700)
CALL ARCI(100.500.40.450.2700)
CALL ARCFI(250.650.80.2700.450)
CALL ARC(250.0.500.0.80.0.2700.450)

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.
CALL COLOR(GREEN)
CALL RECTII(400.600.550.700)
CALL CMOV2I(420.640)
CALL CHARST('Green',5)

CALL COLOR(RED)
CALL RECTFI(600.600.800.650)
CALL COLOR(BLACK)
CALL CMOV2(690.0.620.0)
CALL CHARST('Red',3)

CALL COLOR(BLUE)
CALL RECT(810.0.700.0.1000.0.20.0)
CALL CMOV2I(900.300)
CALL CHARST('Blue',4)

Now draw some text with a ruler on top to measure it by.

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
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)

NOTE: Fortran pads STRING with spaces to its defined length, and LEN(STRING) returns the defined length (50) instead of the length of the character substring inserted into it.

CALL CMOV2I(300,364)
STRING = 'This line is drawn correctly'
STRLEN = LEN('This line is drawn correctly')
CALL CHARST(STRING,STRLEN)

This is the only way (other than counting by hand, as was done for the second part of this line, below) of getting the length of the actual set of characters to be printed.

CALL CHARST('in two parts.',13)

CALL CMOV2I(300,352)
STRLEN = LEN('This line is only 12 pixels lower.')
CALL CHARST('This line is only 12 pixels lower.',STRLEN)

CALL CMOV2I(300,338)
STRLEN = LEN('Now move down 14 pixels ...')
CALL CHARST('Now move down 14 pixels ...',STRLEN)

CALL CMOV2I(300,322)
STRLEN = LEN('And now down 16 ...')
CALL CHARST('And now down 16 ...',STRLEN)

CALL CMOV2I(300,304)
STRLEN = LEN('Now 18 ...')
CALL CHARST('Now 18 ...',STRLEN)

CALL CMOV2I(300,284)
STRLEN = LEN('And finally, 20 pixels.')
CALL CHARST('And finally, 20 pixels.',STRLEN)

Finally, show off the entire font. The cmov2i() before each character is necessary in case that character is not defined.

DO 300 I=0,3
   DO 200 J=0,31
      CALL CMOV2I(300 + 9*I, 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(I:1) = CHAR(32*I + J)
CALL CHARST(ONECH,1)
400 CONTINUE
500 CONTINUE
C
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 PLOT3(). 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 "fill" stands for filled. The characters "2i" mean the same thing that they do for the move and draw commands: "2i" 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 ARC3() command. As was the case with polygons, the "fill" 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 tenths 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. 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 are defined in the default font. 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.

```c
/* "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 numpts 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 max-
imum values and have things work out, but this will make every integer fall ex-
actly 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 re-
peatedly 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 ex-
changes 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:

```
#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);
    }
```
```c
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:

    while(getbutton(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:

```c
/* 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;
        }
    }
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 */
        exit();
        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:

```c
/* "draw.c" */
#include "gl.h"
#include "device.h"

main()
{
    Device val, xpos, ypos;
    ginit();
    color(BLACK);
    curseoff();
    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 */
                gezit();
                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 cursoroff() and cursorn() 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: Popup Menus

The program "popup.c" demonstrates how a simple popup 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 popup menu event, and dispatches to the appropriate routine. Depending on which popup entry the user selects, it draws a line, some points, a circle, a filled or unfilled rectangle, or exits.

The popup 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 popup 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 popup menus make use of the color map. Here, we make use of two bitplanes for the popup menu. In this simple example, the main program itself uses only one bitplane — it only draws in black and white, and assuming that no popup menus were used, it could get away with two mapcolor() commands:

```c
mapcolor(0, 0, 0, 0);
mapcolor(1, 255, 255, 255);
```

The color map is set up for the popup menus as follows: Pixels on the screen that are not under the popup 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:

```c
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 < menu-count) — 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:

```c
/* "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();
cursun();
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);
    }
    cursun();
}
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(8);  /* restrict to menu planes */
    color(2);  /* menu background */
    curoff();
    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);
}
cursor();
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);
            rectf(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);
            rectf(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;
cursor();
} else /* the cursor is outside the menu */ {
    if (lasthighlight != -1) {
        cursoff();
        color(2);
        rectf(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);
curoff();
rectfi(menuleft, menubottom, menuright, menutop);
curon();
if (menuleft < x && x < menuright &&
menubottom < y && y < menutop)
    x = (menutop - y)/16;
else
    x = 0;
brake;
}
}

popmatrix(); /* now restore the state to what the user had */
color(savecolor);
writeimage(savemask);
viewport(llx, urx, lly, ury);
return names[x].type;

Here is the FORTRAN version of "popup":

cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
C    Installation note:
C Various Fortran compilers may require different styles of INCLUDEs.
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
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 as per
C    the parameters above, but are easier to 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 CIRC(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
C
END
C
C
INTEGER FUNCTION POPUP(NAMES,STRNGS)
INTEGER NAMES(*)
CHARACTER*20 STRNRS(*)
INTEGER GETCOL, GETWRI, GETVAL, QTEST
INTEGER I, MOUNT, MTOP, MBOTTOM, MLEFT, MRIGHT
INTEGER LASTHL, HIGHLT
INTEGER*2 DUMMY, X, Y
INTEGER SVCOLR, ESVASK
INTEGER*2 LLX, LLY, URX, URY
INTEGER ILLX, ILLY, IURX, IURY
LASTHL = -1
C

Version 2.1
CALL QREAD(DUMMY)
CALL QREAD(X)
CALL QREAD(Y)

C save the state of everything
SVCOLR = GETCOL()
SVMASK = GETWRI()
CALL GETVIE(ILLX, 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

   WTOP = Y + MCOUNT * 8
   IF (WTOP .GT. 767) WTOP = 767
   MBOTOM = WTOP - MCOUNT * 16

C

   IF (MBOTOM .LT. 0) THEN
      MBOTOM = 0
      WTOP = MBOTOM + MCOUNT * 16
   END IF

C

   MLEFT = X - 100
   IF (MLEFT .LT. 0) MLEFT = 0
   WRIGHT = MLEFT + 200

C

   IF (WRIGHT .GT. 1023) THEN
      WRIGHT = 1023
      MLEFT = 823
   END IF

C

C restrict to menu planes
CALL WRITEM(6)
C menu background
CALL COLOR(2)
CALL CURSOR()  
CALL RECTFI(MLEFT, MBOTOM, MRIGHT, MTOP)  
C         menu text  
CALL COLOR(4)  
CALL RECTFI(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(STRINGS(I), LEN(STRINGS(I)))  
2000 CONTINUE  
C  
CALL CURSOR()  
C         loop to highlight potential selection & accept selection  
3000 CONTINUE  
    X = GETVAL(266)  
    Y = GETVAL(267)  
C  
    IF ((MLEFT.LT.X).AND.(X.LT.MRIGHT) .AND. (MBOTOM.LT.Y) .AND. (Y.LT  
2 .MTOP)) THEN  
        HIGHLT = (MTOP - Y)/16  
        CALL CURSOR()  
C  
    IF ((LASTHL.NE.-1) .AND. (LASTHL.NE.HIGHLT)) THEN  
        CALL COLOR(2)  
        CALL RECTFI(MLEFT+1, MTOP - LASTHL*16 - 15, MRIGHT-1, MTOP -  
2 LASTHL*16 -1)  
        CALL COLOR(4)  
        CALL CMOV2I(MLEFT + 10, MTOP - 14 - LASTHL*16)  
        CALL CHARST(STRINGS(LASTHL+1), LEN(STRINGS(LASTHL+1)))  
    END IF  
C  
    IF (LASTHL.NE.HIGHLT) THEN  
        CALL COLOR(8)  
        CALL RECTFI(MLEFT+1, MTOP - HIGHLT*16 - 15, MRIGHT-1, MTOP -  
2 HIGHLT*16 -1)  
        CALL COLOR(4)  
        CALL CMOV2I(MLEFT + 10, MTOP - 14 - HIGHLT*16)  
        CALL CHARST(STRINGS(HIGHLT+1), LEN(STRINGS(HIGHLT+1)))  
    END IF  
C  
    LASTHL = HIGHLT  
    CALL CURSOR()  
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, MRIGHT-1, MTOP - 2 * 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. (MBOTOM.LT.Y) .AND. (Y. 2LT.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 WITEM(SVMASK)
    CALL VIEWPO(ILXLX, ILRX, ILLY, IURY)
    POPUP = NAMES(X)
    GO TO 4000
END IF

C C loop back to highlight potential selection & accept selection
    GO TO 3000
C
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 */
}<do forever>
```

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 speci-
fied 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:

```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 ROTOY 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(0xffff);
    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, 0.0, 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();
    exit();
    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(100 + i*200, 44);
    sprintf(str, "%6.2f", minval + i*(maxval - minval)/4.0);
    charstr(str);
}
curon();
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()
    {

    IRIS User's Guide
register i;

mapcolor(BLACKDRAW, 0, 0, 0); /* background only */
mapcolor(GREENDRAW, 0, 255, 0); /* green drawing */
mapcolor(REDDRRAW, 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;
}

cursoff();
rectfi(menuleft, menubottom, menuright, menutop);
colr(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(0);
cmov2i(menuleft + 10, menutop - 14 - highlight*16);
charstr(names[highlight].text);
}
lasthighlight = highlight;
curson();
} else /* the cursor is outside the menu */ {
if (lasthighlight != -1) {
curoff();
color(4);
rectfi(menuleft+1, menutop - lasthighlight*16 - 15,
menuright-1, menutop - lasthighlight*16 - 1);
color(0);
cmov2i(menuleft + 10, menutop - 14 - lasthighlight*16);
charstr(names[lasthighlight].text);
curson();
lasthighlight = -1;
}
if (qtest()) {
qread(&dummy);
qread(&x);
qread(&y);
color(0);
curoff();
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 brilliant 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:

<table>
<thead>
<tr>
<th>BP3 (cut)</th>
<th>BP2 (poly)</th>
<th>BP1 (diff)</th>
<th>BP0 (metal)</th>
<th>RED</th>
<th>GREEN</th>
<th>BLUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>255</td>
<td>255</td>
<td>255</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>255</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>150</td>
<td>255</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>255</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>150</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>255</td>
<td>255</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>150</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
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:

```c
/* "vlsi.c" */
#include "gl.h"
#include "device.h"

main()
{
    register i;
    Device dummy, xend, yend, xstart, ystart, type;
    short wn;

    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) {
                        cursoroff();
                        if (type == LEFTMOUSE)
                            color(31); /* draw */
                        else
                            color(0); /* erase */
                        rectf(xstart, ystart, xend, yend);
                        cursen();
                        gflush();
                    }
                }
            }
}  

restart()
{
    writemask(0xffff);
    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);
    cursur();
}

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, and 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:

```c
/* 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;

    glinit();
    color(0);
    writemask(0xffff); /* 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, 0xffff);
    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));
blackscreen(FALSE);
}
displaymap()
{
    register i, j;
    static initialized = 0;
    char redstr[10], greenstr[10], bluestr[10];

    if (!initialized)
    {
        makeobj(i);
        color(MYBLACK);
        clear();
        for (i = 0; i < 3; i++)
        {
            for (j = 0; j < 64; j++) {
                color(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);
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 mapcolor0 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.
INTRODUCTION


Each description defines the number, order, and types of the arguments to each command. C, FORTRAN, and Pascal descriptions are provided under the heading SPECIFICATION. A description of the command, including function, side effects, and potential errors is given in the section called DESCRIPTION; related commands are listed under SEE ALSO.

Some of the commands come in several versions, depending on the number and type of the arguments. Coordinate data can be 2D or 3D, and can be floating-point numbers, integers, or short (16-bit) integers. The default is 3D floating-point data. Integer data and 2D points are specified with suffixes: "i" for integer, "s" for short, and "2" for 2D.

An icon appears on each page to indicate whether the command can be compiled into a display list:

These commands can be used in immediate mode or compiled into a display list.

These commands can be used only in immediate mode.

TEXT STRINGS

C and Pascal text strings are terminated with a null character (ASCII 0); FORTRAN has a character data type that includes the length of the string.

POinters

Many of the Graphics Library commands return several values to the caller. The arguments to these commands are pointers, or addresses of memory locations. In C, they are declared as pointer variables by prefixing the variable name with an asterisk in the declaration. FORTRAN passes all parameters by reference, so no special declaration is necessary. Pascal uses the var keyword to distinguish between value parameters and pointer arguments.

PASCAL ARRAYS

Pascal normally copies all arguments to a subroutine, including arrays. In the interest of efficiency, we have declared all array arguments to be reference parameters: an address to the data will be passed to the subroutine, rather than the data itself.
Furthermore, the Pascal language forces the size of the array to be part of the type of the array. Many Graphics Library commands accept variable-length arrays. Therefore, we have defined oversized arrays with MAXARRAY entries. The user can use some part of the array, or redefine MAXARRAY to a more realistic value.

**BOOLEANS**

Many of the commands have boolean arguments or return boolean values. These are declared as type Boolean in C and Pascal, and as logical or integer in FORTRAN. We assume that FALSE is zero, and that TRUE is anything except FALSE.

**TYPE DECLARATIONS**

We have constructed type declarations for C and Pascal wherever they add to the readability of the code. Here are the type definitions:

```
C
    #define FALSE 0
    #define TRUE !FALSE
    #define PATTERN_16 16
    #define PATTERN_32 32
    #define PATTERN_64 64
    #define PATTERN_16_SIZE 16
    #define PATTERN_32_SIZE 64
    #define PATTERN_64_SIZE 256

typedef unsigned char Byte;
typedef long Boolean;
typedef short Angle;
typedef short Scoord;
typedef short Screencoord;
typedef long Icoord;
typedef float Coord;
typedef char *String;
typedef float Matrix[4][4];

typedef unsigned short Device;

typedef unsigned short Colorindex;
typedef unsigned char RGBvalue;

typedef unsigned short Linestyle;
typedef unsigned short Cursor[16];
typedef struct {
    unsigned short offset;/* 2 bytes */
    Byte w,h; /* 2 bytes */
    char xoff,yoff; /* 2 bytes */
    short width; /* 2 bytes */
```
typedef long Object;
typedef long Tag;

typedef unsigned short Pattern16[PATTERN_16_SIZE];
typedef unsigned short Pattern32[PATTERN_32_SIZE];
typedef unsigned short Pattern64[PATTERN_64_SIZE];

Pascal

const MAXARRAY = 1023;
MAXRASTER = 4095;
MAXOBJECTS = 4095;
PATTERN_16 = 16;
PATTERN_32 = 32;
PATTERN_64 = 64;
PATTERN_16_SIZE = 16;
PATTERN_32_SIZE = 64;
PATTERN_64_SIZE = 256;

type Byte = 0..255;
Short = integer;
UnsignedShort = 0..65535;
Angle = integer;
Screencoord = short;
Icoord = longint;
Coord = real;
Scoord = UnsignedShort;
String = packed array [1..128] of char;
Matrix = array [0..3, 0..3] of real;
Device = UnsignedShort;
string128 = string [128];
pstring = 'String;
pshort = 'Short;
ppshort = 'pshort;

Colorindex = UnsignedShort;
RGBvalue = Byte;

Linestyle = UnsignedShort;
Pattern = array [0..15] of Byte;
Pattern16 = [0..PATTERN_16_SIZE] of UnsignedShort;
Pattern32 = [0..PATTERN_32_SIZE] of UnsignedShort;
Pattern64 = [0..PATTERN_64_SIZE] of UnsignedShort;
Cursor = array [0..15] of UnsignedShort;
Fontchar = record
  offst: Short;
  w, h: Byte;
end;
xoff, yoff: -128..127;
width: Short;
end;

Object = longint;
Tag = longint;

Coord4array = array [0..MAXARRAY, 0..3] of Coord;
Coord3array = array [0..MAXARRAY, 0..2] of Coord;
Coord2array = array [0..MAXARRAY, 0..1] of Coord;
Icoord3array = array [0..MAXARRAY, 0..2] of Icoord;
Icoord2array = array [0..MAXARRAY, 0..1] of Icoord;
Scoord3array = array [0..MAXARRAY, 0..2] of Scoord;
Scoord2array = array [0..MAXARRAY, 0..1] of Scoord;
Screenarray = array [0..MAXARRAY, 0..2] of Screencoord;

Booleanarray = array [0..MAXARRAY] of Boolean;
Colorarray = array [0..MAXARRAY] of Colorindex;
RGBarray = array [0..MAXARRAY] of RGBvalue;

Objarray = array [0..127] of Object;
Fontchararray = array [0..127] of Fontchar;
Font raster = array [0..MAXRASTER] of Short;
Ibuffer = array [0..MAXOBJECTS] of Object;
Queuearray = array [0..MAXARRAY] of Short;
Curvearray = array [0..3] [0..2] of Coord;
Ibuffer = array [0..MAXARRAY] of Short;
Patcharray = array [0..3] [0..3] of Coord;
LIST OF GRAPHICS LIBRARY COMMANDS

arc - draws a circular arc
arcf - draws a filled circular arc
attachcursor - attaches the cursor to two valuators
backbuffer - enables updating in the back buffer
backface - turns on and off back-facing polygon removal
bbox2 - specifies bounding box and minimum pixel radius for drawing commands
blankscreen - turns off screen refresh
blink - changes the color map entry at a selectable rate
blkqread - reads multiple entries from the queue
callfunc - calls a function from within an object
callobj - draws an instance of an object
capture, rcapture - dumps screen images to file or hard copy device
charstr - draws a string of raster characters on the screen
chunksize - specifies minimum object size in memory
circ - outlines a circular region
circf - draws a filled circle
clear - clears the viewport
clearhitcode - sets the system hitcode to zero
cdkoff - turns off the keyboard click
cdkon - turns on the keyboard click
closeobj - closes an object
cmov - changes the current character position
color - sets the color attribute
compactify - compacts the memory storage of an object
crv - draws a curve
crvn - draws a series of curves segments
curorigin - sets the origin of a cursor
curoff - turns the cursor off
curson - turns the cursor on
curvebasis - sets the basis matrix used in drawing curves
curveit - draws a curve segment
curveprecision - sets the number of line segments comprising a curve segment

cyclemap - cycles through color maps at a selected rate
dbtex - sets the dial and switch box text
defbasis - defines a basis matrix
defcursor - defines a cursor glyph
deflinestyle - defines a linestyle
defpattern - defines a texture pattern
defrasterfont - defines a raster font
delobj - deletes an object
deltag - deletes tags from objects
depthcue - turns on and off depth-cue mode
doublebuffer - sets the display mode to double buffer mode
draw - draws a line
editobj - opens an object for editing
endfeedback - turns off feedback mode
endpick - turns off picking mode
endselect - turns off selecting mode
feedback - turns on feedback mode
finish - waits until the terminal command queue and Geometry Pipeline are empty
font - selects a raster font for drawing text strings
foreground - requests that a process not be put in the background by getport while using the window manager
frontbuffer - enables updating in the front buffer
fudge - specifies fudge values to be added to a graphics port when it is resized
gbegin - initializes the IRIS without altering the color map
gconfig - reconfigures the IRIS
genobj - returns a unique integer for use as an object name
genntag - returns a unique integer for use as a tag
getbuffer - indicates which buffers are enabled for writing
getbutton - gets the state (up/down) of a button
getcmmode - returns the current color map mode
getcolor - returns the current color
getcpos - returns the current character position
getcursor - returns the cursor characteristics
getdcm - indicates whether depth-cue mode is on or off
getdepth - returns the parameters of setdepth
getdisplaymode - returns the current display mode
getfont - returns the current raster font number
getgpos - returns the current graphics position
getheight - returns the maximum height of the characters in the current raster font
gethitcode - returns the current system hitcode
getlsbackup - returns the status of the linestyle backup mode
getlsrepeat - returns the linestyle repeat count
getlstyle - returns the current linestyle
getlinewidth - returns the current linewidth
getmap - returns the number of the currently selected color map
getmatrix - returns the current transformation matrix
getmcolor - returns a color map entry
getmem - returns amount of available memory
getmonitor - returns the display monitor mode
getopenobj - indicates if an object is currently open for editing
getorigin - returns the position of a graphics port in the window manager
getpattern - returns the index of the currently selected pattern
getplanes - returns the number of available bitplanes
getport - creates a graphics port in the window manager
getresetls - returns the status of resetls
gRGBcolor - returns the current RGB value
gRGBcursor - returns the characteristics of the cursor in RGB mode
gRGBmask - returns the current RGB writemask
getscrmask - returns the current screenmask
getsize - returns the size of a graphics port in the window manager
gett - returns the location of the current textport
getvaluator - returns the current state of a valuator
getviewport - returns the current viewport
getwritemask - returns the current writemask
getzbuffer - indicates whether z-buffering is on or off
gexit - terminates an IRIS program
gflush - forces all unsent commands down the network to the Geometry Pipeline
ginit - initializes the IRIS
greset - resets all global attributes to their initial values
gsync - waits for a vertical retrace period
imakebackground - makes a process in charge of drawing the background of the window manager screen
initnames - initializes the name stack
isobj - indicates whether a given object name identifies an object
istag - indicates whether a given tag is in use within the currently open object
keepaspect - specifies the aspect ratio of a graphics port
lampoff - turns off display lights on the keyboard
lampon - turns on display lights on the keyboard
linewidth - specifies the linewidth
loadmatrix - loads a transformation matrix
loadname - replaces the name on the top of the name stack
lookat - defines a viewing transformation
lsbackup - controls whether the last two pixels of a line are colored
lsrepeat - sets repeat factor for linestyle
makeobj - creates an object
maketag - names the command in the display list
mapcolor - changes a color map entry
mapw - maps a point on the screen into a line in 3D world coordinates
mapw2 - maps a point on the screen into 2D world coordinates
maxsize - specifies a maximum size for a graphics port in the window manager
minsize - specifies a minimum size for a graphics port in the window manager
move - moves to a specified point
multimap - utilizes the color map as sixteen small maps
multimatrix - pre-multiplies the current transformation matrix
newtag - creates a new tag in an object
noise - filters valuator motion
noport - specifies that a program does not require a graphics port
objdelete - deletes commands from an object
objinsert - inserts commands in an object at the chosen location
objreplacce - overwrites existing display list commands with new ones
onemap - organizes the color map as one large map
ortho - defines an orthographic projection transformation
pagecolor - sets the color of the textport background
pagewritemask - sets the writemask for the textport background
passthrough - passes a single token through the Geometry Pipeline
patch - draws a surface patch
patchbasis - sets current basis matrices
patchcurves - sets number of curves used to represent a patch
patchprecision - sets precision at which curves are drawn
pclos - polygon close
pdr - polygon draw
perspective - defines a perspective projection transformation
pick - puts the system in picking mode
pickszize - sets the dimensions of the picking window
pmv - polygon move
pnt - draws a point
polarview - defines the viewer’s position in polar coordinates
polf - draws a filled polygon
poly - outlines a polygon
popattributes - pops the attribute stack
popmatrix - pops the transformation matrix stack
popname - pops a name off the name stack
popviewport - restores the viewport, screenmask, and setdepth parameters
prefposition - specifies the preferred location and size of a graphics port
prefsize - specifies the preferred size of a graphics port in the window manager
pushattributes - pushes attributes on a stack
pushmatrix - pushes down the transformation matrix stack
pushname - pushes a new name on the name stack
pushviewport - save the current viewport, screenmask, and setdepth parameters
qdevice - queues a device (keyboard, button, or valuator)
qenter - creates an event queue entry
qread - reads the first entry in the event queue
qreset - empties the event queue
qtest - checks the contents of the event queue
rdr - relative draw
readRGB - returns values of specific pixels
readpixels - returns values of specific pixels
rect - outlines a rectangular region
rectcopy - copies a rectangle of pixels on the IRIS screen
rectf - fills a rectangular area
resetls - controls the continuity of linestyles
reshapeviewport - sets the viewport to the current dimensions of the graphics port in the window manager
RGBcolor - sets the current color in RGB mode
RGBcursor - sets the characteristics of the cursor in RGB mode
RGBmode - sets a display mode that bypasses the color map
RGBwritemask - grants write access to a subset of the available bitplanes
ringbell - rings the keyboard bell
rmv - relative move
rotate - rotates graphical primitives
rpdr - relative polygon draw
rpmv - relative polygon move
scale - scales and mirrors objects
screenspace - puts a program in screen space under the window manager
scrmask - defines a clipping mask for the screen
select - puts the IRIS in selecting mode
setbell - sets the duration of the keyboard bell
setcursor - sets the cursor characteristics
setdblights - sets the lights on the dial and button box
setdepth - sets up a 3D viewport
setlinestyle - selects a linestyle
setmap - chooses one of the sixteen small color maps
setmonitor - sets the monitor type
setpattern - selects a texture pattern for filling polygons, rectangles, and curves
setshade - sets the current polygon shade
setvaluator - assigns an initial value to a valuator
shaderange - sets range of color indices
singlebuffer - writes and displays all the bitplanes
spclos - draws currently open polygon
splf - draws a shaded filled polygon
stepunit - specifies that a graphics port should change size in discrete steps
strwidth - returns the width of the specified text string
swapbuffers - swaps the front and back buffers in double buffer mode
swapinterval - defines a minimum time between buffer swaps
textcolor - sets the color of text drawn in the textport
textinit - initializes the textport
textport - allocates an area of the screen for the textport
textwritemask - grants write permission for the textport
tie - ties two valuators to a button
tpoff - turns off the textport
tpon - turns on textport
translate - translates graphical primitives
unqdevice - unqueues a device from the event queue
viewport - allocates an area of the screen for an image
window - defines a perspective projection transformation
writeRGB - paints a row of pixels on the screen
writemask - grants write permission to available bitplanes
writepixels - paints a row of pixels on the screen
xfpt - transforms points
zbuffer - starts or ends z-buffer operation
zclear - initializes the z-buffer to largest possible integer
NAME

arc - draws a circular arc

SPECIFICATION

C
arc(x, y, radius, startang, endang)
Coord x, y, radius;
Angle startang, endang;
arc(x, y, radius, startang, endang)
Icoord x, y, radius;
Angle startang, endang;
arc3(x, y, radius, startang, endang)
Scoord x, y, radius;
Angle startang, endang;

FORTRAN
subroutine arc(x, y, radius, stang, endang)
real x, y, radius
integer*4 stang, endang
subroutine arcI(x, y, radius, stang, endang)
integer*4 x, y, radius, stang, endang
subroutine arc3(x, y, radius, stang, endang)
integer*2 x, y, radius
integer*4 stang, endang

Pascal
procedure arc(x, y, radius: Coord; startang, endang: Angle);
procedure arcI(x, y, radius: Icoord; startang, endang: Angle);
procedure arc3(x, y, radius: Scoord; startang, endang: Angle);

DESCRIPTION

arc draws a circular arc. The arc is defined by a center point, a starting angle, an ending angle, and a radius. The angle is measured from the positive x-axis and specified in integral tenths of degrees. Positive angles describe counterclockwise rotations. Since an arc is a two-dimensional shape, these commands have only 2D forms. The arc is drawn in the x-y plane, with z=0, and uses the current color, linestyle, linewidth and witemask. It is drawn counterclockwise from startang to endang, so that an
arc from 10 degrees to 5 degrees is a nearly complete circle. After the execution of the arc command, the graphics position is undefined.

SEE ALSO

arcf, circ, circf, crv
NAME

arcf - draws a filled circular arc

SPECIFICATION

C

arcf(x, y, radius, startang, endang)
Coord x, y, radius;
Angle startang, endang;
arcfi(x, y, radius, startang, endang)
Icoord x, y, radius;
Angle startang, endang;
arcsf(x, y, radius, startang, endang)
Scoord x, y, radius;
Angle startang, endang;

FORTRAN

subroutine arcf(x, y, radius, stang, endang)
real x, y, radius
integer*4 stang, endang
subroutine arcfi(x, y, radius, stang, endang)
integer*4 x, y, radius, stang, endang
subroutine arcsf(x, y, radius, stang, endang)
integer*2 x, y, radius
integer*4 stang, endang

Pascal

procedure arcf(x, y, radius: Coord; startang, endang: Angle);
procedure arcfi(x, y, radius: Icoord; startang, endang: Angle);
procedure arcsf(x, y, radius: Scoord; startang, endang: Angle);

DESCRIPTION

arcf draws a filled circular arc (pie section). The arc is specified as a center
point, a starting angle, an ending angle, and a radius. The angle is meas-
ured from the x-axis and specified in integral tenths of degrees. Positive
angles describe counterclockwise rotations. The arc is drawn using the
current color and writemask, and filled with the current texture. Since an
arc is a two-dimensional shape, these commands have only 2D forms. The
arc is in the x-y plane with z=0. Arcs are drawn counterclockwise from
starting to endang, so the arc from 10 degrees to 5 degrees is a nearly complete circle.

SEE ALSO
arc, circ, circf, crv
NAME

attachcursor - attaches the cursor to two valuators

SPECIFICATION

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

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

Pascal
procedure attachcursor(vx, vy: Device);

DESCRIPTION

attachcursor attaches the cursor to the movement of two valuators. It takes
two arguments, both valuator device numbers. The first valuator controls
the horizontal motion of the cursor; the second argument determines vertical
motion. The values of the valuators determine the cursor position in screen
coordinates.

SEE ALSO

noise, tie

NOTE

This command can be used only in immediate mode.
NAME

backbuffer - enables updating in the back buffer

SPECIFICATION

C

backbuffer(b)
Boolean b;

FORTRAN

subroutine backbu(b)
logical b

Pascal

procedure backbuffer(b: Boolean);

DESCRIPTION

backbuffer enables updating in the back bitplane buffer. If b = TRUE, the
default, the buffer is enabled; FALSE means it is not enabled. This com-
mand is useful only in double buffer mode, and is ignored in single buffer
and RGB modes.

backbuffer is set to TRUE by gconfig.

SEE ALSO

doublebuffer, frontbuffer, getbuffer
NAME

backface - turns on and off backfacing polygon removal

SPECIFICATION

C

backface(b)
Boolean b;

FORTRAN

subroutine backfa(b)
logical b

Pascal

procedure backface(b: Boolean);

DESCRIPTION

backface initiates or terminates backfacing polygon removal. A backfacing polygon is defined to be 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. The backface utility improves the performance of programs which represent solid objects as collections of polygons. Because the orientation of polygons is determined in screen space, backfacing polygon removal is not a rigorous technique. 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 ALSO

zbuffer
NAME
bbox2 - specifies bounding box and minimum pixel radius for drawing commands

SPECIFICATION

C
bbox2(xmin, ymin, x1, y1, x2, y2)
SCreencoord xmin, ymin;
Coord x1, y1, x2, y2;
bbox2i(xmin, ymin, x1, y1, x2, y2)
SCreencoord xmin, ymin;
Icoord x1, y1, x2, y2;
bbox2s(xmin, ymin, x1, y1, x2, y2)
SCreencoord xmin, ymin;
Scoord x1, y1, x2, y2;

FORTRAN
subroutine bbox2(xmin, ymin, x1, y1, x2, y2)
integer*4 xmin, ymin
real x1, y1, x2, y2
subroutine bbox2i(xmin, ymin, x1, y1, x2, y2)
integer*4 xmin, ymin, x1, y1, x2, y2
subroutine bbox2s(xmin, ymin, x1, y1, x2, y2)
integer*4 xmin, ymin
integer*2 x1, y1, x2, y2

Pascal
procedure bbox2(xmin, ymin: SCreencoord; x1, y1, x2, y2: Coord);
procedure bbox2i(xmin, ymin: SCreencoord; x1, y1, x2, y2: Icoord);
procedure bbox2s(xmin, ymin: SCreencoord; x1, y1, x2, y2: Scoord);

DESCRIPTION
The bbox2 command controls whether the commands in an object are executed. Its arguments are an object space bounding box and minimum horizontal and vertical feature sizes, in pixels. The bounding box is
transformed to screen coordinates and compared with the viewport. If the bounding box is completely outside the viewport, the commands between the `bbox2` command and the end of the object will be ignored. Otherwise, the bounding box of the transformed bounding box is compared with the minimum feature size. If the bounding box is too small in both the x and y dimensions, the rest of the commands in the object are ignored. Otherwise, interpretation of the object continues.

SEE ALSO

`makeobj`
NAME

blankscreen - turns off screen refresh

SPECIFICATION

C

blankscreen(bool)
Boolean bool;

FORTRAN

subroutine blanks(bool)
logical bool

Pascal

procedure blankscreen(bool: Boolean);

DESCRIPTION

blankscreen turns on and off screen refresh. bool = TRUE stops display and bool = FALSE restarts display. When bitplanes are being simultaneously viewed and updated (as in single-buffer mode, RGB mode, and when the front buffer is being displayed in double buffer mode) there is memory contention, and hence reduced performance. This is especially true for non-interlaced monitors. To speed up drawing in these cases, turn the display off while drawing.

NOTE

This command can be used only in immediate mode.
NAME

blink - changes the color map entry at a selectable rate

SPECIFICATION

C

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

FORTRAN

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

Pascal

procedure blink(rate: Short; color: Colorindex; red, green, blue: RGBvalue);

DESCRIPTION

blink specifies a blink rate, a color map index, and red, green, and blue values. Every rate vertical retracts, the color located at index color in the current colormap is updated. Its value is either the original value or the new value supplied by the red, green, and blue arguments.

Up to twenty colors can blink simultaneously, each at a different rate. The blink rate can be changed by calling blink again with the same color and a different rate. Calling blink with rate=0 when color specifies a blinking colormap entry terminates the blinking and restores the original color.

SEE ALSO

mapcolor, color

NOTE

This command can be used only in immediate mode.
NAME

blkqread - reads multiple entries from the queue

SPECIFICATION

C
long blkqread(data, n)
short data[];
short n;

FORTRAN
integer*4 function blkqread(data, n)
integer*2 data(*)
integer*4 n

Pascal
function blkqread(var data: Queuearray, n: Short):longint;

DESCRIPTION

blkqread allows multiple entries to be read from the input queue. This may
be useful to programs running remotely. It is called with an array of shorts
and the length of this array in shorts. blkqread returns the number of
shorts in the array data (i.e., twice the number of event queue entries).

SEE ALSO

qread

NOTE

This command can be used only in immediate mode.
NAME

callfunc - calls a function from within an object

SPECIFICATION

C

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

FORTRAN

NOTE: works only in C

Pascal

NOTE: works only in C

DESCRIPTION

callfunc allows an arbitrary function call from within an object. When it is executed 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.
NAME

callobj - draws an instance of an object

SPECIFICATION

C       callobj(obj)
     Object obj;

FORTRAN subroutine callobj(obj)
     integer*4 obj

Pascal  procedure callobj(obj: Object);

DESCRIPTION

callobj draws an instance of a previously defined object. Its argument is the object name. If callobj specifies an undefined object, the command is ignored.

Global attributes are not saved before a callobj. Thus, if attributes like color are changed within an object, the change may affect the caller as well. Use pushattributes and popattributes to preserve global attributes across callobj commands.

SEE ALSO

makeobj, pushattributes, popattributes
NAME

capture, rcapture - dumps screen images to file or hard copy device

SPECIFICATION

C

capture(dest,cmap)
char *dest;
RGBvalue cmap[][3];
rcapture(dest,cmap,left,right,bottom,top,
        dithsize,dithmat,res)
char *dest;
RGBvalue cmap[][3];
Screeencoord left,right,bottom,top;
long dithsize,res;
short **dithmat;

FORTRAN

subroutine captur(dest,cmap)
character(*) dest
integer*2 cmap(3,*)

subroutine rcaptu(dest,cmap,left,right,bottom,top,
        dithsize,dithmat,res)
character(*) dest
integer*2 cmap(3,*),dithmat(dithsize,dithsize)
integer*4 left,right,bottom,top,dithsize,res

Pascal

procedure capture(dest:String; var cmap:RGBarray);
procedure rcapture(dest:String; var cmap:RGBarray;
       left, right, bottom, top:Scoord;
       dithsize:integer; var dithmat:ppshort; res:integer);

DESCRIPTION

capture dumps a representation of the screen into the file dest in a format compatible with a Tektronix 4692 color inkjet printer. If dest is 0, then it will create a file called capture.img. The function used to open dest for output allows for spooling to other machines, such as print servers (see the coloropen manual pages in Section 3 of the UNIX Programmer's Manual, Volume 1). If cmap is 0, then capture will read the colormap from the hardware. Otherwise, cmap should be a pointer to an array of RGBvalue triplets. The size of
the array depends on how many bitplanes you have and whether you are dumping an RGB image or a colormapped image. For a colormapped image, the size of the array should be 2^N. The array is used as a software colormap of indices that identify the red, green, and blue values for each pixel. For RGBmode, only the first 256 entries in the array are used. In RGBmode, each channel (red, green, and blue) is looked up individually.

rcapture is an extension of capture that allows you to specify a few more variables. The variables left, right, bottom, top determine a rectangular section of the screen to capture. dithsize and dithmat are used for software dithering. dithmat should be a square matrix of size dithsize. If either dithsize or dithmat (or both) are 0, then no dithering is done.

The Tektronix printer accepts data containing 1, 2, or 4 bits per RGB component per pixel. The last argument, res, selects which of these formats to generate. The printer only has four colors (cyan, magenta, yellow, and black) and uses internal dithering if more than one bit per component is specified. One bit per component is used if dithering is selected.

SEE ALSO

Manual pages for the following commands are located in the UNIX Programmer's Manual:

colord(1) coloropen(3) tek(7)

NOTE

This command can be used only in immediate mode.

This command only works on workstations.
NAME

charstr - draws a string of raster characters on the screen

SPECIFICATION

C

charstr(str)
String str;

FORTRAN

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

Pascal

procedure charstr(str: pstring);

DESCRIPTION

charstr draws a string of text using a raster font. The origin of the first character in the string will be placed at the current character position. After each character is drawn, the current character position is updated by the character's spacing parameter. The text string is drawn in the currently selected raster font and color, using the current writemask. Characters that are not defined in the current raster font are ignored.

In FORTRAN, there are two arguments to charstr: str is the name of the text string and length is the number of characters in that string.

SEE ALSO

cmov, defrasterfont, font
NAME

chunksize - specifies minimum object size in memory

SPECIFICATION

C
chunksize(chunk)
long chunk;

FORTRAN
subroutine chunks(chunk)
integer*4 chunk

Pascal
procedure chunksize(chunk: longint);

DESCRIPTION

chunksize specifies the minimum object size. It should be called only once
after ginit and before the first makeobj. As objects grow, they grow in units
of size chunk. This command should be used only if there is a severe
memory shortage. The default chunk size is 1020 bytes.

SEE ALSO

compactify, ginit

NOTE

This command can be used only in immediate mode.
NAME
circ - outlines a circular region

SPECIFICATION

C

circ(x, y, radius)
Coord x, y, radius;
circi(x, y, radius)
Icoord x, y, radius;
circs(x, y, radius)
Scoord x, y, radius;

FORTRAN

subroutine circ(x, y, radius)
real x, y, radius
subroutine circi(x, y, radius)
integer*4 x, y, radius
subroutine circs(x, y, radius)
integer*2 x, y, radius

Pascal

procedure circ(x, y, radius: Coord);
procedure circi(x, y, radius: Icoord);
procedure circs(x, y, radius: Scoord);

DESCRIPTION

circ outlines a circle. The circle has its center at point (x,y) and a radius radius, both specified in world coordinates. Since a circle is two-dimensional, these commands have only 2D forms. The circle is in the x-y plane, with z=0, and is drawn using the current color, linestyle, linewidth, and writemask.

SEE ALSO

arc, arcf, circf, crv
NAME

circf - draws a filled circle

SPECIFICATION

C

circf(x, y, radius)
Coord x, y, radius;
circfi(x, y, radius)
Icoord x, y, radius;
circfs(x, y, radius)
Scoord x, y, radius;

FORTRAN

subroutine circf(x, y, radius)
real x, y, radius
subroutine circfi(x, y, radius)
integer*4 x, y, radius
subroutine circfs(x, y, radius)
integer*2 x, y, radius

Pascal

procedure circf(x, y, radius: Coord);
procedure circfi(x, y, radius: Icoord);
procedure circfs(x, y, radius: Scoord);

DESCRIPTION

circf draws a filled circle, using the current color, writemask, and pattern. The circle has its center at (x,y) and a radius radius, both specified in world coordinates. Since a circle is a two-dimensional shape, these commands have only 2D forms. The circle is drawn in the x-y plane, with z=0.

SEE ALSO

arc, arcf, circ, crv
NAME
  clear - clears the viewport

SPECIFICATION
  C
    clear()
  FORTRAN
    subroutine clear
  Pascal
    procedure clear;

DESCRIPTION
  The clear command sets the screen area within the current viewport to the current color using the current writemask and pattern.

SEE ALSO
  rect, rectf
clearhitcode

clearhitcode

NAME

clearhitcode - sets the system hitcode to zero

SPECIFICATION

C               clearhitcode()

FORTRAN        subroutine clearh

Pascal         procedure clearhitcode;

DESCRIPTION

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

SEE ALSO

gethitcode, pick, select

NOTE

This command can be used only in immediate mode.
NAME
clkoff - turns off the keyboard click

SPECIFICATION
C
clkoff()

FORTRAN
subroutine clkoff

Pascal
procedure clkoff;

DESCRIPTION
clkoff turns off the keyboard click.

SEE ALSO
clkon, lampoff, lampon, ringbell, setbell

NOTE
This command can be used only in immediate mode.
NAME

clkon - turns on the keyboard click

SPECIFICATION

C

clkon()

FORTRAN

subroutine clkon

Pascal

procedure clkon;

DESCRIPTION

clkon turns on the keyboard click.

SEE ALSO

clkoff, lampoff, lampon, ringbell, setbell

NOTE

This command can be used only in immediate mode.
NAME

closeobj - closes an object

SPECIFICATION

C

closeobj()

FORTRAN

subroutine closeo

Pascal

procedure closeobj;

DESCRIPTION

closeobj closes an open object. A new object is created and opened with
makeobj. All display list commands between makeobj and closeobj become
part of the object definition. An existing object is opened for editing with
editobj; the editing session is terminated when the object is closed with a
closeobj.

If no object is open, closeobj is ignored.

SEE ALSO

editobj, makeobj

NOTE

This command can be used only in immediate mode.
cmov

NAME

cmov - changes the current character position

SPECIFICATION

C

cmov(x, y, z)
Coord x, y, z;
cmovi(x, y, z)
Icoord x, y, z;
cmovs(x, y, z)
Scoord x, y, z;
cmov2(x, y)
Coord x, y;
cmov2i(x, y)
Icoord x, y;
cmov2s(x, y)
Scoord x, y;

FORTRAN

subroutine cmov(x, y, z)
real x, y, z
subroutine cmovi(x, y, z)
integer*4 x, y, z
subroutine cmovs(x, y, z)
integer*2 x, y, z
subroutine cmov2(x, y)
real x, y
subroutine cmov2i(x, y)
integer*4 x, y
subroutine cmov2s(x, y)
integer*2 x, y

Pascal

procedure cmov(x, y, z: Coord);
procedure cmovi(x, y, z: Icoord);
procedure cmovs(x, y, z: Scoord);
procedure cmov2(x, y: Coord);
procedure cmov2i(x, y: Icoord);
procedure cmov2s(x, y: Scoord);

DESCRIPTION

`cmov` is analogous to `move` except that it updates the current character position. Its arguments specify a world space point, and can be integers, shorts, or real numbers in 2D or 3D space. This world space point is transformed; the resulting screen space point becomes the new character position. The character position is undefined if the transformed point is outside the viewport.

`cmov` leaves the current graphics position undefined.

SEE ALSO

charstr, move, readpixels, readRGB, writepixels, writeRGB
NAME

color - sets the color attribute

SPECIFICATION

C

    color(c)
    Colorindex c;

FORTRAN

    subroutine color(c)
    integer*4 c

Pascal

    procedure color(c: Colorindex);

DESCRIPTION

color sets the current color which is used until another color command changes it. The current color is an index into a color map. In onemap mode, the color can be in the range from 0 to 4095. In multimap mode, it is used in conjunction with a color map number and should be between 0 and 255. The color is also bounded by the number of available planes.

SEE ALSO

getchr, RGBcolor, RGBwritemask, writemask
NAME
compactify - compacts the memory storage of an object

SPECIFICATION
C
compactify(obj)
Object obj;

FORTRAN
subroutine compac(obj)
integer*4 obj

Pascal
procedure compactify(obj: Object);

DESCRIPTION
As an open object is modified by the various editing commands, it may be stored inefficiently in memory. When the amount of wasted space becomes too large, compactify is called automatically during the closeobj operation. compactify is available to the user to perform the compaction explicitly. Unless insertion of some sort is done, compaction is never necessary. Since it is somewhat expensive, it should not be called unless storage space is critical.

SEE ALSO
closeobj, chunksize

NOTE
This command can be used only in immediate mode.
NAME

**crv** - draws a curve

SPECIFICATION

C

```c
crv(geom)
Coord geom[4][3];
```

FORTRAN

```fortran
subroutine crv(geom)
real geom(3,4)
```

Pascal

```pascal
procedure crv(geom: Curvarray);
```

DESCRIPTION

**crv** draws a curve segment using the current curve basis and precision. The shape of the curve segment is determined by the four control points specified in `geom`.

SEE ALSO

`curvebasis`, `defbasis`, `curveprecision`, `crvn`
NAME

crvn - draws a series of curves segments

SPECIFICATION

C

    crvn(n, geom)
    long n;
    Coord geom[3];

FORTRAN

    subroutine crvn(n, geom)
    integer*4 n
    real geom(3,n)

Pascal

    procedure crvn(n: longint; geom: Coord3array);

DESCRIPTION

crvn draws a series of curve segments using the current basis and precision. The shapes of the curve segments are determined by the control points specified in geom. If the current basis is a B-spline, Cardinal spline or basis of similar properties the curve segments will be joined end to end and will appear as a single curve.

SEE ALSO

curvebasis, curveprecision, crv, defbasis
NAME

curorigin - sets the origin of a cursor

SPECIFICATION

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

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

Pascal
procedure curorigin(n, xorigin, yorigin: Short);

DESCRIPTION

curorigin sets the origin of a cursor. The origin is the position on the cursor that will be aligned with the current cursor valuators. The lower left corner of the cursor has coordinates (0,0). The cursor must already be defined with defcursor when curorigin is called. n is an index into the cursor table created by calls to defcursor.

SEE ALSO

defcursor, attachcursor

NOTE

This command can be used only in immediate mode.
NAME

cursoff - turns the cursor off

SPECIFICATION

C       cursoff()

FORTRAN subroutine cursoff

Pascal  procedure cursoff;

DESCRIPTION

cursoff prevents the cursor from being displayed. This command should
precede any drawing commands that write into the currently displayed cur-
sor bitplanes.

By default, the cursor is always displayed. Before it is drawn on the screen,
the image that it covers is saved away. When the cursor moves, the saved
image is restored. If the image changes while the cursor is displayed, the
saved image may no longer be valid. This is a concern in single buffer and
RGB modes and in double buffer mode when the front buffer is enabled.

SEE ALSO

cursor, getcursor, setcursor

NOTE

This command can be used only in immediate mode.
NAME
curson - turns the cursor on

SPECIFICATION
C   curson()

FORTRAN  subroutine curson

Pascal  procedure curson;

DESCRIPTION
cursorn causes the system to update and display a cursor automatically. cursorn commands are usually paired with cursoff commands, though there is no harm in calling cursorn when the automatic cursor is already visible.

SEE ALSO
attachcursor, cursoff, getcursor, setcursor

NOTE
This command can be used only in immediate mode.
NAME

*curvebasis* - sets the basis matrix used in drawing curves

SPECIFICATION

C

```
curvebasis(basisid)
short basisid;
```

FORTRAN

```
subroutine curveb(basisid)
integer*4 basisid
```

Pascal

```
procedure curvebasis(basisid: integer);
```

DESCRIPTION

*curvebasis* selects a basis matrix (defined by *defbasis*) to be the current basis matrix for drawing curve segments.

SEE ALSO

*crv, crvn, curveprecision, defbasis*
NAME  

curveit - draws a curve segment

SPECIFICATION

C       
curveit(iterationcount)  
   short iterationcount;

FORTRAN subroutine curveit(count)  
   integer*4 count

Pascal  
   procedure curveit(iterationcount: Short);

DESCRIPTION

curveit iterates the forward differences of the matrix on top of the transformation matrix stack iterationcount times, issuing a draw command with each iteration. This command provides access to the low-level hardware capabilities for curve drawing.

SEE ALSO

crv, resetls
NAME

curveprecision - sets the number of line segments comprising a curve segment

SPECIFICATION

C
   curveprecision(nsegments)
   short nsegments;

FORTRAN
   subroutine curvecp(nsegments)
   integer*4 nsegments

Pascal
   procedure curveprecision(nsegments: integer);

DESCRIPTION

curveprecision sets the number of line segments used in drawing curves. Whenever crv or crvn is executed, each curve segment is approximated by nsegments straight line segments. The greater the value of nsegments the smoother the curve.

SEE ALSO

crv, curvebasis, crvn
NAME

cyclemap - cycles through color maps at a selected rate

SPECIFICATION

C

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

FORTRAN

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

Pascal

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

DESCRIPTION

cyclemap specifies a duration (in vertical retraces), a related map, and what map to change to when that duration has timed out. For example, the following commands 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 that program termination does not stop these commands; you must explicitly set all durations to 0.

SEE ALSO

blink, multimap, gconfig

NOTE

This command can be used only in immediate mode.
NAME

dbtext - sets the dial and switch box text

SPECIFICATION

C

dbtext(str)
char str[8];

FORTRAN

subroutine dbtext(str)
character*(8) str

Pascal

procedure dbtext(str: pstring);

DESCRIPTION

dbtext places up to eight characters of text into the text window on the dial and switch box. Only digits, spaces, and upper-case letters are allowed.

NOTE

This command can be used only in immediate mode.
defbasis

NAME
defbasis - defines a basis matrix

SPECIFICATION

C
defbasis(id, mat)
short id;
Matrix mat;

FORTRAN
subroutine defbas(id, mat)
integer*4 id
real mat(4,4)

Pascal
procedure defbasis(id: integer; mat: Matrix);

DESCRIPTION
defbasis allows the user to define basis matrices for use in the generation of curves and patches. matrix is saved and is associated with id. id may then be used in subsequent calls to curvebasis and patchbasis.

SEE ALSO
curvebasis, curveprecision, patchbasis, patchprecision, patchcurves, crv, crvn, patch

NOTE
This command can be used only in immediate mode.
defcursor

NAME
defcursor - defines a cursor glyph

SPECIFICATION

C

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

FORTRAN

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

Pascal

```pascal
procedure defcursor(n: Short; curs: Cursor);
```

DESCRIPTION
defcursor defines a cursor glyph. The arguments are a table index and a 16x16 bitmap. By default, the origin of the cursor is at the lower left corner, but it can be reset with the curorigin command. The cursor origin is the position influenced by valuators attached to the cursor, and is also the position used by pick for the picking region. The index into the table is used as the cursor character's name in subsequent cursor commands. By default, an arrow is defined as cursor 0, and cannot be redefined. To replace a cursor, define the new one to have the same index as the old one.

SEE ALSO
deflinestyle, defrasterfont, defpattern, curorigin, getcursor, pick, setcursor

NOTE
This command can be used only in immediate mode.
NAME

deflinestyle - defines a linestyle

SPECIFICATION

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

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

Pascal  procedure deflinestyle(n: Short; ls: Linestyle);

DESCRIPTION

deflinestyle defines a linestyle, a write-enable pattern that is applied when
lines are drawn. The arguments specify an index into a table where the
linestyles are stored, and a sixteen-bit pattern. Up to $2^{16}$ linestyles can be
defined. By default, index 0 contains the pattern FFFF, which draws solid
lines, and cannot be redefined. There is no performance penalty for drawing
non-solid lines. To replace a linestyle, define the new one to have the same
index as the old one.

SEE ALSO

defcursor, defrasterfont, defpattern, getlstyle, setlinestyle

NOTE

This command can be used only in immediate mode.
NAME

defpattern - defines a texture pattern

SPECIFICATION

C
  defpattern(n, size, mask)
  short n, size;
  Ushort mask[];

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

Pascal
  procedure defpattern(n: longint; size: Short; mask: Ibuffer);

DESCRIPTION

defpattern defines an arbitrary texture pattern. The first argument specifies
an index into a table of patterns. The next, the size of the pattern: 16, 32, or
64 for a 16 x 16, 32 x 32, or 64 x 64 bit pattern, respectively. Finally, the last
argument is an array of shorts that form the actual bit pattern. The pattern
is described from left to right, bottom to top – in the same way as characters
in a raster font. By default, pattern 0 is a 16 x 16 solid pattern and cannot be
changed. There is no performance penalty for non-solid patterns.

SEE ALSO

defcursor, defrasterfont, deflinestyle, getpattern, setpattern

NOTE

This command can be used only in immediate mode.
NAME
defrasterfont - defines a raster font

SPECIFICATION

C

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

FORTRAN

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

Pascal

procedure defrasterfont(n, ht, nc, nr: Short; var chars: Fntchrarray;
var raster: Fontraster);

DESCRIPTION
defrasterfont defines a raster font and has six arguments. The first two are
n, an index into the font table, and ht, an integer specifying the maximum
height, in pixels, of characters in the font. The index n becomes the font’s
internal name. nc gives the number of characters in the font and thus 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 this character’s bounding box, an offset into the array of rasters,
and the amount to add to the current character position x after drawing the
character. raster is an array of nr shorts of bitmap information. It is a one-
dimensional array of mask shorts, ordered left to right and then bottom to
top. Mask bits are left-justified in the character’s bounding box.

To replace a raster font, define the new one to have the same index as the
old one. To delete a raster font, define a font with no characters. The
default font, 0, is a fixed pitch font of height 16 and width 9. Font 0 cannot
be redefined.

SEE ALSO
charstr, font

NOTE

This command can be used only in immediate mode.
NAME

delobj - deletes an object

SPECIFICATION

C
    delobj(obj)
    Object obj;

FORTRAN
    subroutine delobj(obj)
    integer*4 obj

Pascal
    procedure delobj(obj: Object);

DESCRIPTION

delobj deletes an object. Most of the display list storage associated with the object is freed, and the object name is marked undefined until it is used to create a new object. Attempts to call an object with the deleted name will be ignored.

SEE ALSO

makeobj, compactify

NOTE

This command can be used only in immediate mode.
NAME

deltag - deletes tags from objects

SPECIFICATION

C       deltag(t)
       Tag t;

FORTRAN subroutine deltag(t)
      integer*4 t

Pascal   procedure deltag(t: Tag);

DESCRIPTION

deltag removes the tag t from the object currently open for editing. The special tags STARTTAG and ENDTAG cannot be deleted.

SEE ALSO

editobj, maketag

NOTE

This command can be used only in immediate mode.
NAME

depthcue - turns on and off depth-cue mode

SPECIFICATION

C
    depthcue(mode)
    Boolean mode;

FORTRAN
    subroutine depthc(mode)
    logical mode

Pascal
    procedure depthcue(mode: Boolean);

DESCRIPTION

depthcue sets the current depth cue mode. If mode is TRUE all line and
points drawn will be drawn depth cued. This means that the color of the
lines and points is determined from their z values and the range of color
indices specified by shaderange. The z values, whose range is set by the set-
depth command, are mapped linearly into the range of color indices. In this
mode, lines with large delta z’s will span the full range of colors, typically
appearing bright at one end and dim at the other.

In order for depth cueing to work properly the color map locations in the
range specified by shaderange must be loaded with a series of colors gradu-
ally increasing or decreasing in intensity.

SEE ALSO

setdepth, shaderange
NAME

doublebuffer - sets the display mode to double buffer mode

SPECIFICATION

C

doublebuffer()

FORTRAN

 subroutine double

Pascal

 procedure doublebuffer;

DESCRIPTION

The IRIS has three display modes: single buffer, double buffer, and RGB. doublebuffer sets the display mode to double buffer mode. It does not take effect until after a call to gconfig. In double buffer mode, the bitplanes are partitioned into two groups, called the front and back planes. Only the front planes are displayed. Drawing commands normally update only the back planes; frontbuffer and backbuffer can override the default.

gconfig sets frontbuffer = TRUE and backbuffer = FALSE in double buffer mode.

SEE ALSO

backbuffer, frontbuffer, gconfig, getbuffer, getdisplaymode, RGBmode, singlebuffer, swapbuffers

NOTE

This command can be used only in immediate mode.
NAME

draw - draws a line

SPECIFICATION

C

draw(x, y, z)
Coord x, y, z;
drawi(x, y, z)
Icoord x, y, z;
draws(x, y, z)
Scoord x, y, z;
draw2(x, y)
Coord x, y;
draw2i(x, y)
Icoord x, y;
draw2s(x, y)
Scoord x, y;

FORTRAN

subroutine draw(x, y, z)
real x, y, z
subroutine drawi(x, y, z)
integer*4 x, y, z
subroutine draws(x, y, z)
integer*2 x, y, z
subroutine draw2(x, y)
real x, y
subroutine draw2i(x, y)
integer*4 x, y
subroutine draw2s(x, y)
integer*2 x, y

Pascal

procedure draw(x, y, z: Coord);
procedure drawi(x, y, z: Icoord);
procedure draws(x, y, z: Scoord);
procedure draw2(x, y: Coord);
procedure draw2i(x, y: Icoord);
procedure draw2s(x, y: Scoord);

DESCRIPTION

draw connects the specified point and the current graphics position with a line segment using the current linestyle, linewidth, color (if in depth cue mode, then the depth cued color is used), and writemask. The current graphics position is updated to the specified point. Commands that invalidate the current graphics position should not be placed within sequences of moves and draws.

SEE ALSO

move, pnt, rdr, rmv
NAME
editobj - opens an object for editing

SPECIFICATION

C
editobj(obj)
Object obj;

FORTRAN
subroutine editobj(obj)
integer*4 obj

Pascal
procedure editobj(obj: Object);

DESCRIPTION
editobj opens an object for editing. A command pointer acts as a cursor for appending new commands. The command pointer is initially set to the end of the object. Graphics commands are appended to the object until either a closeobj or a pointer positioning command (objdelete, objinsert, or objreplace) is executed. Usually, the user need not be concerned about storage allocation. Objects grow and shrink automatically as commands are added and deleted.

If editobj specifies an undefined object, an error message is printed.

SEE ALSO
compactify, objdelete, objinsert, objreplace

NOTE
This command can be used only in immediate mode.
NAME

endfeedback - turns off feedback mode

SPECIFICATION

C
long endfeedback(buffer)
short buffer[ ];

FORTRAN
integer*4 function endfee(buffer)
integer*2 buffer(*)

Pascal
function endfeedback(buffer: Short): longint;

DESCRIPTION

dendfeedback turns off feedback mode. buffer contains the values output by the Geometry Pipeline during the feedback session. The value returned by endfeedback is the number of shorts in buffer.

SEE ALSO

feedback

NOTE

This command can be used only in immediate mode.
endpick

NAME

delpick - turns off picking mode

SPECIFICATION

C
long endpick(buffer)
short buffer[];

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

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

DESCRIPTION

delpick takes the IRIS out of picking mode. When a drawing command
draws into the picking region, the contents of the name stack are stored in
buffer along with the number of names in the stack. Thus, if the name stack
contained 5, 9, 17 when a hit occurred, the numbers 3, 5, 9, 17 are added to
the buffer. The magnitude of the value returned by endpick is the number
of such name lists in the buffer. If the value returned is positive, then all
hits are recorded in the name lists, if it is negative, the buffer was not large
enough to hold them all.

SEE ALSO

pick, gethitcode, clearhitcode, loadname, initnames

NOTE

This command can be used only in immediate mode.
NAME

endselect - turns off selecting mode

SPECIFICATION

C
long endselect(buffer)
short buffer[];

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

Pascal
function endselect(var buffer: lbuffer): integer;

DESCRIPTION

endselect takes the IRIS out of selecting mode. Any hits caused by drawing commands that occurred between the calls to select and endselect are stored in the buffer. Every hit that occurs causes the entire contents of the name stack to be recorded in the buffer, preceded by the number of names in the stack. Thus, if the name stack contained 5, 9, 17 when a hit occurred, the numbers 3, 5, 9, 17 are added to the buffer. The magnitude of the value returned by endselect is the number of such name lists in the buffer. If the value returned is positive, then all hits are recorded in the name lists, if it is negative, the buffer was not large enough to hold them all.

SEE ALSO

select, gethitcode, clearhitcode, loadname, initnames

NOTE

This command can be used only in immediate mode.
NAME

feedback - turns on feedback mode

SPECIFICATION

C

feedback(buffer, size)
short buffer[];
long size;

FORTRAN

subroutine feedback(buffer, size)
integer*2 buffer(*)
integer*4 size

Pascal

procedure feedback(buffer: lbuffer; size: longint);

DESCRIPTION

The feedback command puts the IRIS in feedback mode. During feedback mode, the output of the Geometry Pipeline is stored in buffer, rather than being sent to the raster display system. size specifies the maximum number of values that can be stored in buffer. While in feedback mode, the IRIS does not draw anything on the screen.

SEE ALSO

defaultfeedback

NOTE

This command can be used only in immediate mode.
NAME

finish - waits until the terminal command queue and Geometry Pipeline are empty

SPECIFICATION

C

    finish()

FORTRAN

    subroutine finish

Pascal

    procedure finish;

DESCRIPTION

finish blocks the host process until all previous commands have been executed. It forces all unsent commands down the Geometry Pipeline to the bitplanes, sends a final token, and blocks until that token has gone through the pipeline and an acknowledgment has been sent. It is useful in the presence of network and pipeline delays.

SEE ALSO

gflush

NOTE

This command can be used only in immediate mode on the IRIS Terminal.
NAME

font - selects a raster font for drawing text strings

SPECIFICATION

C

    font(fn)ntnum)
    short fnnum;

FORTRAN

    subroutine font(fntnum)
    integer*4 fnnum

Pascal

    procedure font(fntnum: Short);

DESCRIPTION

font chooses a raster font that will be used whenever a charstr command
draws a text string. The argument is an index into the font table built with
the defrasterfont command. This font remains in effect until another font
command is executed. By default, font 0 is selected.

If font specifies a font number that is not defined, font 0 is selected.

SEE ALSO

defrasterfont, getfont
NAME

foreground - requests that a process not be put in the background by getport while using the window manager

SYNOPSIS

C       foreground()

FORTRAN subroutine foregr

Pascal  procedure foreground;

DESCRIPTION

Normally, when getport is called, the application is run in the background. If the user wants to interact with the program through the textport, then calling foreground will keep the process in the foreground after getport.

SEE ALSO

getport

NOTE

This command can be used only in immediate mode.
frontbuffer

NAME

frontbuffer - enables updating in the front buffer

SPECIFICATION

C

    frontbuffer(b)
    Boolean b;

FORTRAN

    subroutine frontb(b)
    logical b

Pascal

    procedure frontbuffer(b: Boolean);

DESCRIPTION

frontbuffer enables updating in the front buffer. If \( b = \text{FALSE} \), the default, the front buffer is not enabled; \text{TRUE} means it is enabled. This command is useful only in double buffer mode and is ignored otherwise.

frontbuffer is set to \text{FALSE} by gconfig.

SEE ALSO

backbuffer, doublebuffer, getbuffer
NAME

fudge - specifies fudge values to be added to a graphics port when it is resized

SPECIFICATION

C

fudge(xfudge, yfudge)
long xfudge, yfudge;

FORTRAN

subroutine fudge(xfudge, yfudge)
integer*4 xfudge, yfudge

Pascal

procedure fudge(xfudge, yfudge: longint);

DESCRIPTION

fudge specifies fudge values to be added to the dimensions of a graphics port when it is resized. Typically, this command is used to provide for a window border. The extra area will appear as part of the port and will be reflected in the size of the drag box as the user sizes it. fudge is useful in conjunction with stepunit and keepaspect. With stepunit the port size is constrained to be:

\[
\text{width} = \text{xunit} \times n + \text{xfudge}
\]
\[
\text{height} = \text{yunit} \times m + \text{yfudge}
\]

for integers n and m. With keepaspect the port size is \((w, h)\) where:

\[
(w-\text{xfudge}) \times \text{aspect} = (h-\text{yfudge}) \times \text{aspect}
\]

It is called at the beginning of a graphics program that will be run with the window manager. If getport is not called or if the system is not running the window manager, fudge is ignored.

SEE ALSO

ggetport, keepaspect, stepunit

NOTE

This command can be used only in immediate mode.
NAME

gbegin - initializes the IRIS without altering the color map

SPECIFICATION

C

gbegin()

FORTRAN

subroutine gbegin

Pascal

procedure gbegin;

DESCRIPTION

gbegin may be called instead of ginit as the first Graphics Library command in IRIS programs. gbegin does a ginit without changing the contents of any of the entries in the color map and without calling setcursor.

SEE ALSO

ginit, greset

NOTE

This command can be used only in immediate mode.
NAME

gconfig - reconfigures the IRIS

SPECIFICATION

C           gconfig()

FORTRAN     subroutine gconfi

Pascal      procedure gconfig;

DESCRIPTION

gconfig uses the map mode, display mode, and number of requested planes
to compute and allocate available planes and define the mapping from colors
to bitplanes. doublebuffer, multimap, onemap, RGBmode, setplanes, and
singlebuffer do not take effect until gconfig is called. After the gconfig call,
the writemask, color, cursor color, and writemask are all undefined. The
contents of the color map are unchanged.

SEE ALSO

doublebuffer, multimap, onemap, RGBmode, setplanes, singlebuffer

NOTE

This command can be used only in immediate mode.
NAME

genobj - returns a unique integer for use as an object name

SPECIFICATION

C

Object genobj()

FORTRAN

integer*4 function genobj()

Pascal

function genobj: Object;

DESCRIPTION

Object names can be up to 31 bits long and must be unique within the program. genobj generates unique 31-bit integer names. Names might be pointers into data structures, hashed string names or randomly chosen numbers. genobj will not generate an object name that is currently in use. The user must take care in generating names when using a combination of user-defined and genobj-defined names.

Use of genobj is not required to name an object.

SEE ALSO

callobj, gentag, isobj, makeobj

NOTE

This command can be used only in immediate mode.
NAME

gentag - returns a unique integer for use as a tag

SPECIFICATION

C
Tag gentag()

FORTRAN
integer*4 function gentag()

Pascal
function gentag: Tag;

DESCRIPTION

gentag generates a unique integer to be used as a tag. Tags must be unique within the enclosing object. The gentag command provides a unique 31-bit integer for use when the tag has no significance beyond labeling a command. Note that the generated tag is unique when gentag is executed. If the user later defines a tag with the same value, the first command tag is lost.

Use of gentag is not required to name an object.

SEE ALSO

genobj, istag

NOTE

This command can be used only in immediate mode.
NAME

getbuffer - indicates which buffers are enabled for writing

SPECIFICATION

C       long getbuffer()

FORTRAN  integer*4 function getbuf()

Pascal  function getbuffer: longint;

DESCRIPTION

getbuffer indicates which buffers are enabled for writing. 1, the default, means that 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 the IRIS is not in double buffer mode.

<table>
<thead>
<tr>
<th>Value</th>
<th>Buffer Enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>neither</td>
</tr>
<tr>
<td>1</td>
<td>back buffer</td>
</tr>
<tr>
<td>2</td>
<td>front buffer</td>
</tr>
<tr>
<td>3</td>
<td>both buffers</td>
</tr>
</tbody>
</table>

SEE ALSO

backbuffer, doublebuffer, frontbuffer

NOTE

This command can be used only in immediate mode.
getbutton

NAME

getbutton - gets the state (up/down) of a button

SPECIFICATION

<table>
<thead>
<tr>
<th>Language</th>
<th>Description</th>
</tr>
</thead>
</table>
| C        | `Boolean getbutton(num)`  
           | `Device num;` |
| FORTRAN  | `logical function getbut(num)`  
           | `integer*4 num` |
| Pascal   | `function getbutton(num: Device): longint;` |

DESCRIPTION

getbutton returns the state of the button numbered num. A return value of 0 means that the button is up; 1 means that it is down.

NOTE

The command can be used only in immediate mode.
getcmmode

NAME

getcmmode - returns the current color map mode

SPECIFICATION

C

Boolean getcmmode()

FORTRAN

logical function getcmm()

Pascal

function getcmmode: longint;

DESCRIPTION

getcmmode returns the current color map mode. The returned value is either 0 for multimap mode or 1 for onemap mode.

SEE ALSO

multimap, onemap

NOTE

This command can be used only in immediate mode.
NAME

getchor - returns the current color

SPECIFICATION

C          long getcolor()

FORTRAN    integer*4 function getcol()

Pascal     function getcolor: Colorindex;

DESCRIPTION

getchor returns the current color. It is an index into the color map, and is
meaningful in either single or double buffer mode. The command is ignored
in RGB mode.

SEE ALSO

color, singlebuffer, doublebuffer

NOTE

This command can be used only in immediate mode.
NAME

getcpos - returns the current character position

SPECIFICATION

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

FORTRAN
subroutine getcpos(ix, iy)
integer*2 ix, iy

Pascal
procedure getcpos(var ix, iy: Screencoord);

DESCRIPTION

getcpos returns the current character position (in screen coordinates).

SEE ALSO

getcpos

NOTE

This command can be used only in immediate mode.
NAME

getcursor - returns the cursor characteristics

SPECIFICATION

C

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

FORTRAN

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

Pascal

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

DESCRIPTION

getcursor returns the index of the cursor glyph, the color and the writemask associated with the cursor, and a boolean indicating whether the cursor is automatically displayed and updated by the system. The arguments to the command are addresses of four locations where the four cursor attributes are to be returned. The default is the glyph at location 0 in the cursor table, displayed with the color 1, drawn in the first available plane, and automatically updated and displayed on each vertical retrace. This command is undefined in RGB mode.

SEE ALSO

defcursor, setcursor

NOTE

This command can be used only in immediate mode.
NAME

getdcm - indicates whether depth-cue mode is on or off

SPECIFICATION

C

Boolean getdcm()

FORTRAN

logical function getdcm()

Pascal

function getdcm: longint;

DESCRIPTION

getdcm returns TRUE (0) if the IRIS is in depth-cue mode, and FALSE (1) otherwise.

NOTE

This command can be used only in immediate mode.
NAME

getdepth - returns the parameters of setdepth

SPECIFICATION

C            getdepth(near, far)
            Screencoord *near, *far;

FORTRAN      subroutine getdep(near, far)
            integer*2 near, far

Pascal       procedure getdepth(var near, far: Screencoord);

DESCRIPTION

getdepth returns the near and far parameters of the setdepth command.

SEE ALSO

setdepth

NOTE

This command can be used only in immediate mode.
NAME
getdisplaymode - returns the current display mode

SPECIFICATION
C
long getdisplaymode()

FORTRAN
integer*4 function getdis()

Pascal
function getdisplaymode: longint;

DESCRIPTION
getdisplaymode returns the current display mode, where the values 0, 1, and 2 indicate the display modes as shown:

<table>
<thead>
<tr>
<th>Value</th>
<th>Display Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RGB mode</td>
</tr>
<tr>
<td>1</td>
<td>single buffer mode</td>
</tr>
<tr>
<td>2</td>
<td>double buffer mode</td>
</tr>
</tbody>
</table>

SEE ALSO
doublebuffer, RGBmode, singlebuffer

NOTE
This command can be used only in immediate mode.
NAME
   getfont - returns the current raster font number

SPECIFICATION
   C    long getfont()
   FORTRAN    integer*4 function getfon()
   Pascal    function getfont: longint;

DESCRIPTION
   getfont returns the index of the currently selected raster font.

SEE ALSO
   defrasterfont, font

NOTE
   This command can be used only in immediate mode.
NAME

getgpos - returns the current graphics position

SPECIFICATION

C

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

FORTRAN

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

Pascal

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

DESCRIPTION

getgpos returns the current graphics position.

SEE ALSO

getcpos

NOTE

This command can be used only in immediate mode.
getheight

NAME

getheight - returns the maximum height of the characters in the current raster font

SPECIFICATION

C

long getheight()

FORTRAN

integer*4 function gethei()

Pascal

function getheigh: longint;

DESCRIPTION

getheight returns the maximum height (as defined in the defrasterfont command that defined the font) of the characters in the current raster font, including ascenders (present in characters like 't' and 'h') and descenders (as in 'v' and 'p'). The height is usually the number of pixels between the top of the tallest ascender and the bottom of the lowest descender.

SEE ALSO

strwidth

NOTE

This command can be used only in immediate mode.
NAME

gethitcode - returns the current system hitcode

SPECIFICATION

C

long gethitcode()

FORTRAN

integer*4 function gethit()

Pascal

function gethitcode: longint;

DESCRIPTION

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 six-bit number, as shown below, with one bit for each clipping plane. (Systems configured without near-far clipping always return 0's for the near and far clipping planes.)

<table>
<thead>
<tr>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>far</td>
<td>near</td>
<td>top</td>
<td>bottom</td>
<td>right</td>
<td>left</td>
</tr>
</tbody>
</table>

SEE ALSO

clearhitcode, pick, select

NOTE

This command can be used only in immediate mode.
NAME

getlsbackup - returns the status of the linestyle backup mode

SPECIFICATION

C

Boolean getlsbackup()

FORTRAN

logical function getlsb()

Pascal

function getlsbackup: longint;

DESCRIPTION

getlsbackup returns the current value of the linestyle backup flag. TRUE means that lsbackup is on and the last three pixels of a line will be colored regardless of the linestyle pattern; FALSE means that lsbackup is off and the pattern will determine the state of all the pixels in the line.

SEE ALSO

lsbackup

NOTE

This command can be used only in immediate mode.
NAME

getlsrepeat - returns the linestyle repeat count

SPECIFICATION

C

    long getlsrepeat()

FORTRAN

    integer*4 function getlsr()

Pascal

    function getlsrepeat: longint;

DESCRIPTION

getlsrepeat returns the current linestyle repeat factor (set by lsrepeat).

SEE ALSO

lsrepeat

NOTE

This command can be used only in immediate mode.
NAME

get linestyle - returns the current linestyle

SPECIFICATION

C          long getlinestyle()

FORTRAN    integer*4 function getlst()

Pascal     function get linestyle: longint;

DESCRIPTION

get linestyle returns the currently selected linestyle. The returned value is an
index into the linestyle table.

SEE ALSO

def linestyle, set linestyle

NOTE

This command can be used only in immediate mode.
NAME

getwidth - returns the current linewidth

SPECIFICATION

C

long getwidth()

FORTRAN

integer*4 function getlwi()

Pascal

function getwidth: longint;

DESCRIPTION

getwidth returns the current linewidth in pixels.

SEE ALSO

linewidth

NOTE

This command can be used only in immediate mode.
NAME

getmap - returns the number of the currently selected color map

SPECIFICATION

C       long getmap()

FORTRAN  integer*4 function getmap()

Pascal  function getmap: longint;

DESCRIPTION

getmap returns the number, from 0 to 15, of the currently selected color map. If called in onemap mode, it returns zero.

SEE ALSO

multimap, onemap

NOTE

This command can be used only in immediate mode.
NAME

getmatrix - returns the current transformation matrix

SPECIFICATION

C

getmatrix(m)
Matrix m;

FORTRAN

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

Pascal

procedure getmatrix(var m: Matrix);

DESCRIPTION

getmatrix copies the transformation matrix from the top of the stack to an array provided by the user. The matrix stack is unchanged.

SEE ALSO

loadmatrix, multmatrix

NOTE

This command can be used only in immediate mode.
NAME

getmcolor - returns a color map entry

SPECIFICATION

C

getmcolor(color, r, g, b)
Colorindex color;
short *r, *g, *b;

FORTRAN

subroutine getmco(color, r, g, b)
integer*4 color
integer*2 r, g, b

Pascal

procedure getmcolor(color: Colorindex; var r, g, b: RGBvalue);

DESCRIPTION.

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

SEE ALSO

mapcolor

NOTE

This command can be used only in immediate mode.
NAME

getmem - returns amount of available memory

SPECIFICATION

C
   long getmem()

FORTRAN
   integer*4 function getmem()

Pascal
   function getmem: longint;

DESCRIPTION

getmem returns the amount of available memory left on the system. On a
workstation with virtual memory up to 14 megabytes, it will return 14 MB
less the amount used. On a terminal, it will return the amount of free physi-
cal memory.

NOTE

This command can be used only in immediate mode.
NAME

getmonitor - returns the display monitor mode

SPECIFICATION

C
long getmonitor()

FORTRAN
integer*4 function getmon()

Pascal
function getmonitor: longint;

DESCRIPTION

getmonitor returns the current display monitor mode. Current possibilities include 60Hz non-interlaced, 30Hz interlaced, and NTSC. In the file get.h, values for HZ30, HZ60, and NTSC are defined.

NOTE

This command can be used only in immediate mode.
NAME

`getopenobj` - indicates if an object is currently open for editing

SPECIFICATION

C

```c
Object getopenobj()
```

FORTRAN

```fortran
integer*4 function getope()
```

Pascal

```pascal
function getopenobj: Object;
```

DESCRIPTION

`getopenobj` returns the number of the object currently open for editing. If no object is open, it returns -1. It is useful for writing routines that create objects, but may be called while another object is open for editing.

NOTE

This command can be used only in immediate mode.
NAME

getorigin - returns the position of a graphics port in the window manager

SPECIFICATION

C

getorigin(x, y)
long *x, *y;

FORTRAN

subroutine getori(x, y)
integer*4 x, y

Pascal

procedure getorigin(x, y: longint);

DESCRIPTION

getorigin returns the position of the lower left corner of a graphics port. It should only be called after getport. When the window manager is not running, getorigin always returns (0, 0).

SEE ALSO

getport

NOTE

This command can be used only in immediate mode.
NAME

getpattern - returns the index of the currently selected pattern

SPECIFICATION

C

long getpattern()

FORTRAN

integer*4 function getpat()

Pascal

function getpattern: longint;

DESCRIPTION

getpattern returns the index of the pattern currently in use from the table of available patterns.

SEE ALSO

defpattern, setpattern

NOTE

This command can be used only in immediate mode.
NAME
getplanes - returns the number of available bitplanes

SPECIFICATION
C
   long getplanes()

FORTRAN
   integer*4 function getpla()

Pascal
   function getplanes: longint;

DESCRIPTION
getplanes returns the number of available bitplanes in the system.

SEE ALSO
   multimap, onemap

NOTE
   This command can be used only in immediate mode.
NAME

getport - creates a graphics port in the window manager

SPECIFICATION

C

getport(name)
char name[];

FORTRAN

subroutine getpor(name)
character*(*) name

Pascal

procedure getport(name: pstring);

DESCRIPTION

getport does a graphics initialization and creates a graphics port based on
the characteristics specified in the commands preceding the call to getport. If
no characteristics are specified or the description is incomplete, the window
manager prompts the user for the missing information. The user employs
the cursor to show the size and location of the graphics port. getport puts
the graphics program in the background, unless the foreground command is
called (before getport).

When it is called outside the window manager, getport creates a standard-
sized graphics port: (xmin 0, xmax 1023, ymin 0, ymax 767).

SEE ALSO

minsize, maxsize, keepaspect, prefsizer, prefposition, stepunit, fudge,
noport, foreground

NOTE

This command can be used only in immediate mode.
NAME

gresetls - returns the status of resetls

SPECIFICATION

C

Boolean getresetls()

FORTRAN

logical function getres()

Pascal

function getresetls: longint;

DESCRIPTION

gresetls returns the current value of the reset linestyle flag. TRUE, the default value, means that the linestyle will be reinitialized for each line segment. FALSE means that the pattern will rotate continuously across line segment boundaries.

SEE ALSO

resetls

NOTE

This command can be used only in immediate mode.
NAME

getscrmask - returns the current screenmask

SPECIFICATION

C

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

FORTRAN

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

Pascal

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

DESCRIPTION

getscrmask returns the screen coordinates of the current screenmask.

SEE ALSO

scrmask, popviewport, pushviewport

NOTE

This command can be used only in immediate mode.
NAME

**getsize** - returns the size of a graphics port in the window manager

SPECIFICATION

C

```c
getsize(x, y)
long *x, *y;
```

FORTRAN

```fortran
subroutine getsiz(x, y)
integer*4 x, y
```

Pascal

```pascal
procedure getsize(x, y: longint);
```

DESCRIPTION

getsize returns the dimensions (in pixels) of the graphics port being used by a graphics program. It should be called after getport.

If getsize is called outside the window manager, the size will always be (1024, 768).

SEE ALSO

getport

NOTE

This command can be used only in immediate mode.
NAME

 gettext - returns the location of the current textport

SPECIFICATION

C               gettext(left, right, bottom, top)
                Screencoord *left, *right, *bottom, *top;

FORTRAN         subroutine gettext(left, right, bottom, top)
                integer*2 left, right, bottom, top

Pascal          procedure gettext(var left, right, bottom, top: Screencoord);

DESCRIPTION

 gettext returns the location of the current textport.

SEE ALSO

 textport

NOTE

 This command can be used only in immediate mode
NAME

getvaluato - returns the current state of a valuator

SPECIFICATION

C
long getvaluato(val)
Device val;

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

Pascal
function getvaluato(val: Device): longint;

DESCRIPTION

This command returns the current value of the valuator numbered \texttt{val}.

SEE ALSO

getbutton, qvaluator

NOTE

This command can be used only in immediate mode
NAME

**getviewport** - returns the current viewport

SPECIFICATION

C

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

FORTRAN

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

Pascal

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

DESCRIPTION

**getviewport** returns the current viewport, non-destructively reading the top of the viewport stack. The arguments to **getviewport** are the addresses of four memory locations. These will be assigned the left, right, bottom, and top coordinates of the viewport.

SEE ALSO

popviewport, pushviewport, viewport

NOTE

This command can be used only in immediate mode.
NAME

getwritemask - returns the current writemask

SPECIFICATION

C

long getwritemask()

FORTRAN

integer*4 function getwri()

Pascal

function getwritemask: Colorindex;

DESCRIPTION

getwritemask returns the current writemask. It is an integer with up to
twelve significant bits, one for each available bitplane. This command is
undefined in RGB mode.

SEE ALSO

RGBwritemask, writemask

NOTE

This command can be used only in immediate mode.
NAME

getbuffer - indicates whether z-buffering is on or off

SPECIFICATION

C

Boolean getbuffer()

FORTRAN

logical function getbu()

Pascal

function getbuffer: longint;

DESCRIPTION

getbuffer returns the current value of the z-buffer flag. FALSE, the default value, means that z-buffering is off. TRUE means that z-buffering is on.

SEE ALSO

zbuffer, zclear, setdepth

NOTE

This command can be used only in immediate mode.
NAME

gexit - terminates an IRIS program

SPECIFICATION

C       gexit()

FORTRAN subroutine gexit

Pascal  procedure gexit;

DESCRIPTION

gexit is the final graphics command in an IRIS program. It flushes communication buffers and waits for the Geometry Pipeline to empty.

SEE ALSO

ginit, greset, finish

NOTE

This command can be used only in immediate mode.
NAME

gflush - forces all unsent commands down the network to the Geometry Pipeline

SPECIFICATION

C

gflush()

FORTRAN

subroutine gflush

Pascal

procedure gflush;

DESCRIPTION

Most graphics commands are buffered at the host by the communication software for efficient block transfer of data from the host to the IRIS. gflush delivers all buffered yet untransmitted graphics data to the IRIS. Certain graphics commands (notably those which return values) flush the host buffer when they are executed. gflush is only used on an IRIS Terminal.

SEE ALSO

finish

NOTE

This command can be used only in immediate mode.
NAME

ginit - initializes the IRIS

SPECIFICATION

C

ginit()

FORTRAN

subroutine ginit

Pascal

procedure ginit;

DESCRIPTION

ginit is the first command in every IRIS graphics program. It initializes the
hardware, allocates memory for symbol tables and display lists, and sets up
default values for global state variables in the same manner as greset. It has
no arguments and should be called only once, before any other IRIS Graph-
ics Library command.

SEE ALSO

gexit, greset, gbegn

NOTE

This command can be used only in immediate mode.
NAME

greset - resets all global attributes to their initial values

SPECIFICATION

C

greset()  

FORTRAN  subroutine greset

Pascal  procedure greset;

DESCRIPTION

greset returns the following global state variables to their initial values, and can be called at any time.

<table>
<thead>
<tr>
<th>State Variable</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>available bitplanes</td>
<td>all bitplanes(^1)</td>
</tr>
<tr>
<td>backface mode</td>
<td>off</td>
</tr>
<tr>
<td>color</td>
<td>undefined</td>
</tr>
<tr>
<td>color map mode</td>
<td>one map(^2)</td>
</tr>
<tr>
<td>cursor</td>
<td>0 (arrow)(^2)</td>
</tr>
<tr>
<td>depthcue mode</td>
<td>off</td>
</tr>
<tr>
<td>display mode</td>
<td>single buffer</td>
</tr>
<tr>
<td>font</td>
<td>0(^3)</td>
</tr>
<tr>
<td>linestyle</td>
<td>0 (solid)</td>
</tr>
<tr>
<td>linestyle backup</td>
<td>off</td>
</tr>
<tr>
<td>linewidth</td>
<td>1 pixel</td>
</tr>
<tr>
<td>lsrepeat</td>
<td>1</td>
</tr>
<tr>
<td>picking size</td>
<td>10(\times)10 pixels</td>
</tr>
<tr>
<td>reset linestyle</td>
<td>on</td>
</tr>
<tr>
<td>RGB color</td>
<td>undefined</td>
</tr>
<tr>
<td>RGB writemask</td>
<td>undefined</td>
</tr>
<tr>
<td>shaderange</td>
<td>(0,7,0,1023)</td>
</tr>
<tr>
<td>texture</td>
<td>0 (solid)</td>
</tr>
<tr>
<td>viewport</td>
<td>entire screen</td>
</tr>
<tr>
<td>writemask</td>
<td>all planes enabled(^1)</td>
</tr>
<tr>
<td>zbuffer mode</td>
<td>off</td>
</tr>
</tbody>
</table>

\(^1\) Available bitplanes refer to the number of bitplanes used for bitmapped images.
\(^2\) Arrow reference depends on the current cursor position.
\(^3\) Font reference depends on the current font.
1. If there are more than three bitplane boards installed, the number of available planes is set to twelve.
2. The color and writemask of the cursor are set to 1.
3. Rasterfont 0 is a Helvetica-like font.

In addition, `greset` puts a two-dimensional orthographic projection transformation on the matrix stack with left, right, bottom, and top set to the boundaries of the screen. It also turns the cursor on, ties it to `MOUSEX` and `MOUSEY`, and unqueues all buttons, valuators, and the keyboard. Each button is set to `FALSE` and untied from valuators. Each valuator is set to `XMAXSCREEN/2`; the range is `0..XMAXSCREEN`. `MOUSEY` is an exception. It is set to `YMAXSCREEN/2` and has range `0..YMAXSCREEN`.

`greset` also defines every entry in the color map, as follows:

<table>
<thead>
<tr>
<th>Index</th>
<th>Name</th>
<th>RGB Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Red</td>
</tr>
<tr>
<td>0</td>
<td>BLACK</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>RED</td>
<td>255</td>
</tr>
<tr>
<td>2</td>
<td>GREEN</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>YELLOW</td>
<td>255</td>
</tr>
<tr>
<td>4</td>
<td>BLUE</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>MAGENTA</td>
<td>255</td>
</tr>
<tr>
<td>6</td>
<td>CYAN</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>WHITE</td>
<td>255</td>
</tr>
<tr>
<td>all others</td>
<td>unnamed</td>
<td>undefined</td>
</tr>
</tbody>
</table>

**SEE ALSO**

ginit, gbegin

**NOTE**

This command can be used only in immediate mode.
NAME
gRGBcolor - returns the current RGB value

SPECIFICATION

C
   gRGBcolor(red, green, blue)
   short *red, *green, *blue;

FORTRAN
   subroutine gRGBco(red, green, blue)
   integer*2 red, green, blue

Pascal
   procedure gRGBcolor(var red, green, blue: RGBvalue);

DESCRIPTION

gRGBcolor returns the current RGB color values. The arguments are addresses of three locations that will be filled with the red, green, and blue values. The system must be in RGB mode when this command is executed.

SEE ALSO

RGBcolor, RGBmode

NOTE

This command can be used only in immediate mode.
NAME

gRGBcursor - returns the characteristics of the cursor in RGB mode

SPECIFICATION

C

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

FORTRAN

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

Pascal

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

DESCRIPTION

gRGBcursor returns the seven parameters of the last RGBcursor command executed. These are: index, red, green, blue, redm, greenm and bluem. gRGBcursor also returns a boolean b that is TRUE if the automatic cursor is on. The system must be in RGB mode when this command is executed.

SEE ALSO

RGBcursor

NOTE

This command can be used only in immediate mode.
NAME

gRGBmask - returns the current RGB writemask

SPECIFICATION

C

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

FORTRAN

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

Pascal

procedure gRGBmask(var redm, greenm, bluem: RGBvalue);

DESCRIPTION

gRGBmask returns the current RGB writemask as three eight-bit masks. The masks are placed in the locations addressed by redm, greenm and bluem. The system must be in RGB mode when this command is executed.

SEE ALSO

getwritemask, RGBwritemask

NOTE

This command can be used only in immediate mode.
gsync

NAME

    gsync - waits for a vertical retrace period

SPECIFICATION

    C         gsync()

    FORTRAN   subroutine gsync

    Pascal    procedure gsync;

DESCRIPTION

    In single buffer and RGB modes, rapidly changing scenes should be syn-
    chronized with the refresh rate. The gsync command waits for the next verti-
    cal retrace period.

SEE ALSO

    RGBmode, singlebuffer

NOTE

    This command can be used only in immediate mode.
NAME

imakebackground  - makes a process in charge of drawing the background of the window manager screen

SPECIFICATION

C          imakebackground()

FORTRAN    subroutine imakeb

Pascal     procedure imakebackground;

DESCRIPTION

imakebackground is called to register a process as the maintainer of the screen background. It is called instead of the graphics port characteristic commands and getport. The process draws the background and reads the input queue. Every time the process gets a REDRAW token in the queue, it should redraw the background.

SEE ALSO

getport

NOTE

This command can be used only in immediate mode.
NAME

initnames - initializes the name stack

SPECIFICATION

C

initnames()

FORTRAN

subroutine initna

Pascal

procedure initnames;

DESCRIPTION

initnames clears the name stack for picking and selecting.

SEE ALSO

pick, select
NAME

isobj - indicates whether a given object name identifies an object

SPECIFICATION

C       Boolean isobj(obj)
     Object obj;

FORTRAN logical function isobj(obj)
           integer*4 obj

Pascal    function isobj(obj: Object): longint;

DESCRIPTION

isobj returns TRUE if obj names an object and FALSE if it does not. It will return TRUE any time after the makeobj call for the object.

SEE ALSO

genobj, istag, makeobj

NOTE

This command can be used only in immediate mode.
NAME

istag - indicates whether a given tag is in use within the currently open object

SPECIFICATION

C

    Boolean istag(t)
    Tag t;

FORTRAN

    logical function istag(t)
    integer*4 t

Pascal

    function istag(t: Tag): longint;

DESCRIPTION

istag returns TRUE if t is in use within the currently open object and FALSE if it is not in use. The result is undefined if there is no currently open object.

SEE ALSO

gentag, isobj

NOTE

This command can be used only in immediate mode.
NAME

keepaspect - specifies the aspect ratio of a graphics port

SPECIFICATION

C
keepaspect(x, y)
long x, y;

FORTRAN
subroutine keepas(x, y)
integer x, y

Pascal
procedure keepaspect(x, y: longint);

DESCRIPTION

keepaspect specifies the aspect ratio of a graphics port. It is called at the
beginning of a graphics program that will be run with the window manager.
It takes effect when getport is called. The resulting graphics port will main-
tain the aspect ratio specified in keepaspect, even if its size is changed. For
example, keepaspect(1, 1) will always result in a square graphics port. If
getport is not called or if the system is not running the window manager,
keepaspect will be ignored.

SEE ALSO

getport, fudge

NOTE

This command can be used only in immediate mode.
NAME

lampoff - turns off display lights on the keyboard

SPECIFICATION

C

    lampoff(lamps)
    char lamps;

FORTRAN

    subroutine lampoff(lamps)
    integer*4 lamps

Pascal

    procedure lampoff(lamps: Byte);

DESCRIPTION

lampoff turns off any combination of the four user-controllable lamps on the IRI5 keyboard. The four low-order bits of the argument lamps control lamps 1 through 4.

SEE ALSO

clockoff, clkon, lampon, ringbell, setbell

NOTE

This command can be used only in immediate mode.
NAME

`lampon` - turns on display lights on the keyboard

SPECIFICATION

C

```c
lampon(lamps)
char lamps;
```

FORTRAN

```fortran
subroutine lampon(lamps)
integer*4 lamps
```

Pascal

```pascal
procedure lampon(lamps: Byte);
```

DESCRIPTION

`lampon` turns on any combination of the four user-controllable lamps on the keyboard. The four low-order bits of the argument `lamps` control lamps 1 through 4.

SEE ALSO

`clkon`, `clkoff`, `lampoff`, `ringbell`, `setbell`

NOTE

This command can be used only in immediate mode.
NAME

`linewidth` - specifies the linewidth

SPECIFICATION

C
```
linewidth(n)
short n;
```

FORTRAN
```
subroutine linewi(n)
integer*4 n
```

Pascal
```
procedure linewidth(n: Short);
```

DESCRIPTION

`linewidth` specifies the width of a line. The width is the number of pixels along the screen axis having the least change between the endpoints of the line. The wide line will be centered (as far as possible) about the true line. If `linewidth` is set to \( n = 1 \), `resetls` must be `TRUE`.

SEE ALSO

`lsbackup`, `resetls`, `setlinestyle`
loadmatrix

NAME
   loadmatrix - loads a transformation matrix

SPECIFICATION
   C
   loadmatrix(m)
   Matrix m;

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

   Pascal
   procedure loadmatrix(m: Matrix);

DESCRIPTION
   loadmatrix loads a $4 \times 4$ floating point matrix onto the matrix stack, replacing
   the current top of the stack.

SEE ALSO
   multmatrix, popmatrix, pushmatrix
NAME

loadname - replaces the name on the top of the name stack

SPECIFICATION

C
loadname(name)
short name;

FORTRAN
subroutine loadna(name)
integer*4 name

Pascal
procedure loadname(name: Short);

DESCRIPTION

loadname replaces the top name in the name stack with a new 16-bit integer name. The contents of the name stack are stored in a buffer each time a command causes a "hit" in picking or selecting mode.

SEE ALSO

pick, select
NAME

lookat - defines a viewing transformation

SPECIFICATION

C

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

FORTRAN

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

Pascal

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

DESCRIPTION

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

SEE ALSO

polarview
NAME

lsbackup - controls whether the last two pixels of a line are colored

SPECIFICATION

C
    lsbackup(b)
    Boolean b;

FORTRAN
    subroutine lsback(b)
    logical b

Pascal
    procedure lsbackup(b: Boolean);

DESCRIPTION

lsbackup is one of two commands that modify the application of the linestyle pattern. If enabled, it overrides the current linestyle to guarantee that the final two pixels in a line will be colored. It takes one argument, a boolean. If \( b = TRUE \), backup mode is enabled. resets should also be set to TRUE when backup mode is enabled. FALSE, the default setting, means the linestyle will be used as is, and lines may have invisible endpoints.

SEE ALSO

deflinestyle, getlsbackup, setlinestyle, resets

NAME
  lstrepeat - sets repeat factor for linestyle

SPECIFICATION

C
  lsrepeat(factor)
  long factor;

FORTRAN
  subroutine lsrepe(factor)
  integer*4 factor

Pascal
  procedure lsrepe(factor: longint);

DESCRIPTION

lsrepeat sets a repeat factor for the current linestyle. When a line is being
drawn, pixels are turned on if there is a 1 in the corresponding position of
the linestyle mask. Thus the mask 0x5555 specifies alternate pixels be turned
on (assuming the lsrepeat factor is 1). If the repeat factor is n, then the
0x5555 pattern above would draw a line with n bits on and n bits off, alter-
nately.

SEE ALSO

getlsrepeat, setlinestyle
NAME
makeobj - creates an object

SPECIFICATION
C
makeobj(obj)
Object obj;

FORTRAN
subroutine makeobj(obj)
integer*4 obj

Pascal
procedure makeobj(obj: Object);

DESCRIPTION
Objects are created with the makeobj command. makeobj takes an argument, a 31-bit integer name, that will be associated with the object. If an object with the given name previously existed, that object is deleted. When a makeobj command is executed, the object name is entered into a symbol table and memory is allocated for a display list. Subsequent graphics commands are compiled into the display list instead of being executed.

SEE ALSO
callobj, closeobj, genobj, isobj

NOTE
This command can be used only in immediate mode.
NAME

maketag - names the command in the display list

SPECIFICATION

C
maketag(t)
Tag t;

FORTRAN
subroutine maketa(t)
integer*4 t

Pascal
procedure maketag(t: Tag);

DESCRIPTION

Commands can be explicitly named with the maketag command. The user supplies a 31-bit name that is assigned to the command immediately following the maketag command. The tag is local to that object. The same 31-bit name can be used safely in different objects.

SEE ALSO

gentag, istag
NAME

mapcolor - changes a color map entry

SPECIFICATION

C

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

FORTRAN

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

Pascal

procedure mapcolor(color: Colorindex; red, green,
blue: RGBvalue);

DESCRIPTION

mapcolor changes a single color map entry to the specified RGB value. Its
arguments are a color map index and eight bits each of red, green, and blue
intensities. Pixels written with color color will be displayed with the speci-
fied RGB intensities.

In multimap mode only the currently selected color map can be updated
with mapcolor. Invalid indices are ignored.

SEE ALSO

multimap, onemap, setmap

NOTE

This command can be used only in immediate mode.
NAME

mapw - maps a point on the screen into a line in 3D world coordinates

SPECIFICATION

C

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

FORTRAN

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

Pascal

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

DESCRIPTION

mapw takes a pair of 2D screen coordinates and maps them into 3D world space. Since the z-coordinate is missing from screen space, the point becomes a line in world space. mapw computes the inverse mapping from vobj, a viewing object. A viewing object is a graphical object that contains only viewport, projection, viewing transformation, and modelling commands. A correct mapping from screen to world space requires that the viewing object contain the projection and viewing transformations that mapped the displayed object from world to screen coordinates. The world space line that is computed from (sx, sy) and vobj is returned as two points and stored in the locations addressed by wx1, wy1, wz1 and wx2, wy2, wz2.

SEE ALSO

mapw2

NOTE

This command can be used only in immediate mode.
NAME

mapw2 - maps a point on the screen into 2D world coordinates

SPECIFICATION

C

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

FORTRAN

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

Pascal

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

DESCRIPTION

mapw2 is the two-dimensional version of mapw. vobj is a viewing object containing the viewport, projection, viewing, and modelling transformations that define world space; sx and sy define a screen space point. The corresponding world space coordinates are returned in wx and wy. If the transformation is not 2D, the result is undefined.

SEE ALSO

mapw

NOTE

This command can be used only in immediate mode.
maxsize

NAME

maxsize - specifies a maximum size for a graphics port in the window manager

SPECIFICATION

C

    maxsize(x, y)
    long x, y;

FORTRAN

    subroutine maxsiz(x, y)
    integer*4 x, y

Pascal

    procedure maxsize(x, y: long);

DESCRIPTION

maxsize specifies a maximum size for a graphics port. It is called at the beginning of a graphics program that will be run with the window manager, and takes effect when getport is called. The default maximum size is 1024 pixels wide and 768 pixels high. As the graphics port is reshaped, the window manager will not allow it to become larger than this maximum size. If maxsize is called without getport or if the system is not running the window manager, the command will be ignored.

SEE ALSO

getport, minsize

NOTE

This command can be used only in immediate mode.
NAME

minsize - specifies a minimum size for a graphics port in the window manager

SPECIFICATION

C

minsize(x, y)
long x, y;

FORTRAN

subroutine minsize(x, y)
integer*4 x, y

Pascal

procedure minsize(x, y: long);

DESCRIPTION

minsize specifies a minimum size for a graphics port. It is called at the beginning of a graphics program that will be run with the window manager, and takes effect when getport is called. The default minimum size is 40 pixels wide and 30 pixels high. The port can be reshaped by the user, but the window manager will not allow it to become smaller than this minimum size. minsize has no effect if getport is not called or if the program is not being run with the window manager.

SEE ALSO

getport, maxsize

NOTE

This command can be used only in immediate mode.
NAME
move - moves to a specified point

SPECIFICATION

C
move(x, y, z)
Coord x, y, z;
movei(x, y, z)
Icoord x, y, z;
moves(x, y, z)
Scoord x, y, z;
move2(x, y)
Coord x, y;
move2i(x, y)
Icoord x, y;
move2s(x, y)
Scoord x, y;

FORTRAN
subroutine move(x, y, z)
real x, y, z
subroutine movei(x, y, z)
integer*4 x, y, z
subroutine moves(x, y, z)
integer*2 x, y, z
subroutine move2(x, y)
real x, y
subroutine move2i(x, y)
integer*4 x, y
subroutine move2s(x, y)
integer*2 x, y

Pascal
procedure move(x, y, z: Coord);
procedure movei(x, y, z: Icoord);
procedure moves(x, y, z: Scoord);
procedure move2(x, y: Coord);
procedure move2i(x, y: Icoord);
procedure move2s(x, y: Scoord);

DESCRIPTION
The move command moves (without drawing) the current graphics position
to the specified point. The command has six forms: 3D floating point, 3D
integer, 2D floating point, 2D integer, 3D short integer, and 2D short
integer; move2(x,y) is equivalent to move(x,y,0.0).

SEE ALSO
draw, pnt, rdr, rmv
NAME

multimap - utilizes the color map as sixteen small maps

SPECIFICATION

C

multimap()

FORTRAN

subroutine multim

Pascal

procedure multim;

DESCRIPTION

multimap organizes the color map as sixteen small maps, each with a maximum of 256 RGB entries. The number of entries in each map is $2^p$, where $p$ is the number of available planes. multimap does not take effect until gconfig is called.

SEE ALSO

gconfig, getcmmode, getmap, onemap, setmap, setplanes

NOTE

This command can be used only in immediate mode.
multimatrix

NAME

**multimatrix** - pre-multiplies the current transformation matrix

SPECIFICATION

C

```c
multimatrix(m)
Matrix m;
```

FORTRAN

```fortran
subroutine multim(m)
real m(4,4)
```

Pascal

```pascal
procedure multim(m: Matrix);
```

DESCRIPTION

**multimatrix** pre-multiplies the current top of the transformation stack by the given matrix. If \( T \) is the current matrix, **multimatrix(M)** replaces \( T \) with \( M \cdot T \).

SEE ALSO

- loadmatrix
- popmatrix
- pushmatrix
NAME

newtag - creates a new tag in an object

SPECIFICATION

C
newtag(newtag, oldtag, offset)
Tag newtag, oldtag;
long offset;

FORTRAN
subroutine newtag(newtag, oldtag, offset)
integer*4 newtag, oldtag, offset

Pascal
procedure newtag(newtag, oldtag: Tag; offset: longint);

DESCRIPTION

newtag creates a new tag offset positions beyond oldtag.

SEE ALSO

maketag

NOTE

This command can be used only in immediate mode.
NAME

noise - filters valuator motion

SPECIFICATION

C

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

FORTRAN

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

Pascal

procedure noise(v: Device; delta: Short);

DESCRIPTION

noise determines how often queued valuators will make entries in the event queue. Some valuators are "noisy": an unmoving device may report small fluctuations in value. The noise command allows the user to set a lower limit on what constitutes a move. That is, the value of a noisy valuator v must change by at least delta before the motion is considered significant. For example, noise(v,5) means that valuator v must move at least 5 units before a new queue entry is made.

SEE ALSO

setvaluator

NOTE

This command can be used only in immediate mode.
NAME

noport - specifies that a program does not require a graphics port

SPECIFICATION

C noport()

FORTRAN subroutine noport

Pascal procedure noport;

DESCRIPTION

noport specifies that a graphics program does not need any screen space and therefore does not need a graphics port in the window manager. This is useful for programs which only read or write the color map. This command is called at the beginning of a graphics program; a call to getport is still needed to do a graphics initialization. If getport is not called or if the system is not running the window manager, noport is ignored.

SEE ALSO

getport

NOTE

This command can be used only in immediate mode.
NAME

objdelete - deletes commands from an object

SPECIFICATION

C

    objdelete(tag1, tag2)
    Tag tag1, tag2;

FORTRAN

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

Pascal

    procedure objdelete(tag1, tag2: Tag);

DESCRIPTION

objdelete is an object editing command. It removes commands from an object between tag1 and tag2. The tags remain. Any tags defined between tag1 and tag2 are also deleted.

If no object is open for editing when objdelete is called, the command is ignored. This command leaves the pointer at the end of the object after it takes effect.

SEE ALSO

editobj, objinsert, objreplace

NOTE

This command can be used only in immediate mode.
NAME

objinsert - inserts commands in an object at the chosen location

SPECIFICATION

C              objinsert(t)
              Tag t;

FORTRAN       subroutine objins(t)
              integer*4 t

Pascal         procedure objinsert(t: Tag);

DESCRIPTION

objinsert takes a tag t as an argument, and positions an editing pointer on
the indicated command. Subsequent graphics commands are added follow-
ing the tag. closeobj or another positioning command (objdelete, objinsert,
or objreplace) terminates the insertion.

SEE ALSO

closeobj, objdelete, editobj, objreplace, maketag

NOTE

This command can be used only in immediate mode.
NAME

*objreplac* - overwrites existing display list commands with new ones

SPECIFICATION

C

```c
objcreplace(t)
Tag t;
```

FORTRAN

```fortran
subroutine objrep(t)
integer*4 t
```

Pascal

```pascal
procedure objcplace(t: Tag);
```

DESCRIPTION

*objcreplace* combines the functions of *objinsert* and *objdelete*. It takes a single argument, a tag *t*. Graphics commands that follow *objcplace* overwrite existing commands until a *closeobj*, *objcinsert*, *objdelete*, or *objcplace* command terminates the replacement. *objcplace* requires the new command to be the same length as the one it replaces, which makes replacement operations fast. More general replacement can be done with *objdelete* and *objcinsert* commands. *objcplace* is meant as a quick means of replacing a command with a version of itself containing different arguments.

SEE ALSO

closeobj, objdelete, editobj, objinsert

NOTE

This command can be used only in immediate mode.
NAME

onemap - organizes the color map as one large map

SPECIFICATION

C onemap()

FORTRAN subroutine onemap

Pascal procedure onemap;

DESCRIPTION

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
planes. onemap does not take effect until gconfig is called. The IRIS is ini-
tially in onemap mode.

SEE ALSO

gconfig, getcmmode, multimap, setmap, setplanes

NOTE

This command can be used only in immediate mode.
NAME

ortho - defines an orthographic projection transformation

SPECIFICATION

C
ortho(left, right, bottom, top, near, far)
Coord left, right, bottom, top, near, far;
ortho2(left, right, bottom, top)
Coord left, right, bottom, top;

FORTRAN
subroutine ortho(left, right, bottom, top, near, far)
real left, right, bottom, top, near, far
subroutine ortho2(left, right, bottom, top)
real left, right, bottom, top

Pascal
procedure ortho(left, right, bottom, top, near, far: Coord);
procedure ortho2(left, right, bottom, top: Coord);

DESCRIPTION

ortho specifies a box-shaped enclosure in the eye coordinate system that will
be mapped to the viewport. left, right, bottom, and top are the x and y clip-
ning planes. near and far are distances along the line of sight from the eye
space origin, and can be negative. The z clipping planes are at –near and
–far.

ortho2 defines a 2D clipping rectangle. When ortho2 is used with 3D world
coordinates, the z values are left unchanged.

Both ortho and ortho2 load a matrix onto the transformation stack, overwrit-
ing whatever was there.

SEE ALSO

perspective, window
NAME

pagecolor - sets the color of the textport background

SPECIFICATION

C

    pagecolor(c)
    short c;

FORTRAN

    subroutine pageco(c)
    integer*4 c

Pascal

    procedure pagecolor(c: Colorindex);

DESCRIPTION

pagecolor sets the textport background color.

SEE ALSO

color, pagewritemask, textcolor, textport, textwritemask, tpoff, tpun

NOTE

This command can be used only in immediate mode.
NAME

pagewritemask - sets the writemask for the textport background

SPECIFICATION

C  
pagewritemask(pmask)
short pmask;

FORTRAN  
subroutine pagewr(pmask)
integer*4 pmask

Pascal  
procedure pagewritemask(pmask: Colorindex);

DESCRIPTION

pagewritemask sets the textport background writemask.

SEE ALSO

color, pagecolor, textcolor, textport, textwritemask, tpo, tpon, writemask

NOTE

This command can be used only in immediate mode.
NAME

**passthrough** - passes a single token through the Geometry Pipeline

SPECIFICATION

C

    passthrough(token)
    short token;

FORTRAN

    subroutine passth(token)
    integer*4 token

Pascal

    procedure passthrough(token: Short);

DESCRIPTION

**passthrough** passes a single 16-bit integer through the Geometry Pipeline. It is useful in feedback mode to parse the information returned. Suppose, for example, that points are being transformed and clipped by the geometry engines. If **passthrough** is used between every pair of points, then if a point is clipped out, two **passthrough** commands will appear in a row in the output buffer. This command is very dangerous to use when not in feedback mode – it has the effect of sending a random command to the raster subsystem.

NOTE

Should be used only in feedback mode.
NAME

patch - draws a surface patch

SPECIFICATION

C

patch(geomx, geomy, geomz)
Matrix geomx, geomy, geomz;

FORTRAN

subroutine patch(geomx, geomy, geomz)
real geomx(4,4), geomy(4,4), geomz(4,4)

Pascal

procedure patch(geomx, geomy, geomz: Patcharray);

DESCRIPTION

patch draws a surface patch using the current patchbasis, patchprecision, and patchcurves. The shape of the patch is determined by the control points specified in geomx, geomy, and geomz.

SEE ALSO

patchbasis, patchprecision, patchcurves, defbasis
NAME

patchbasis - sets current basis matrices

SPECIFICATION

C

    patchbasis(uid, vid)
    long uid, vid;

FORTRAN

    subroutine patchb(uid, vid)
    integer*4 uid, vid

Pascal

    procedure patchbasis(uid, vid: longint);

DESCRIPTION

patchbasis sets the current basis matrices (defined by defbasis) for both the u and v parametric directions of a surface patch. The current u and v bases are used when the patch command is issued.

SEE ALSO

defbasis, patch, patchprecision, patchcurves
patchcurves

NAME

patchcurves - sets number of curves used to represent a patch

SPECIFICATION

C

    patchcurves(ucurves, vcurves)
    long ucurves, vcurves;

FORTRAN

    subroutine patchc(ucurves, vcurves)
    integer*4 ucurves, vcurves

Pascal

    procedure patchcurves(ucurves, vcurves: long);

DESCRIPTION

patchcurves sets the current number of \( u \) and \( v \) curves used to represent a patch as a wire frame.

SEE ALSO

patch, patchbasis, patchprecision
NAME

patchprecision - sets precision at which curves are drawn

SPECIFICATION

C

    patchprecision(usegments, vsegments)
    long usegments, vsegments;

FORTRAN

    subroutine patchp(usegments, vsegments)
    integer*4 usegments, vsegments

Pascal

    procedure patchprecision(usegments, vsegments: long);

DESCRIPTION

patchprecision sets the precision at which curves defining a wire frame patch are drawn. Precisions are specified independently in the u and v directions for a patch. Patch precisions are similar to curve precisions – they specify the minimum number of line segments used in drawing a patch. The actual number of line segments used in the display of a patch is an even multiple of the number of curves used in drawing the patch greater than the specified precision. This multiple is used in order to guarantee the intersection of patch curves.

SEE ALSO

* curveprecision, patchcurves, patchbasis, patch
NAME

pclos - polygon close

SPECIFICATION

C

pclos()

FORTRAN

subroutine pclos

Pascal

procedure pclos;

DESCRIPTION

pclos closes a filled polygon. It is used to terminate a sequence of pmv and
pdr or rpmv and rpdr commands. The polygon so defined is filled using the
current texture, color, writemask. For example, the following sequence
draws a square:

pmv(0.0, 0.0, 0.0);
pdr(1.0, 0.0, 0.0);
pdr(1.0, 1.0, 0.0);
pdr(0.0, 1.0, 0.0);
pclos();

The results are undefined if the polygon is not convex.

SEE ALSO

pdr, pmv, rpdr, rpmv
NAME

pdr - polygon draw

SPECIFICATION

C

pdr(x, y, z)
Coord x, y, z;
pdri(x, y, z)
Icoord x, y, z;
pdrs(x, y, z)
Scoord x, y, z;
pdr2(x, y)
Coord x, y;
pdr2i(x, y)
Icoord x, y;
pdr2s(x, y)
Scoord x, y;

FORTRAN

subroutine pdr(x, y, z)
real x, y, z
subroutine pdri(x, y, z)
integer*4 x, y, z
subroutine pdrs(x, y, z)
integer*2 x, y, z
subroutine pdr2(x, y)
real x, y
subroutine pdr2i(x, y)
integer*4 x, y
subroutine pdr2s(x, y)
integer*2 x, y

Pascal

procedure pdr(x, y, z: Coord);
procedure pdri(x, y, z: Icoord);
procedure pdrs(x, y, z: Scoord);
procedure pdr2(x, y: Coord);
procedure pdr2i(x, y: lcoord);
procedure pdr2s(x, y: scoord);

DESCRIPTION

pdr is the draw command for filled polygons. A typical polygon is drawn with a pmv, a sequence of pdr's, and is closed with a pclos. For example, the following sequence draws a square:

    pmv(0.0, 0.0, 0.0);
pdr(1.0, 0.0, 0.0);
pdr(1.0, 1.0, 0.0);
pdr(0.0, 1.0, 0.0);
    pclos();

The results are undefined if the polygon is not convex.

SEE ALSO

pclos, pmv, rpdr, rpmv
perspective

NAME

perspective - defines a perspective projection transformation

SPECIFICATION

C

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

FORTRAN

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

Pascal

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

DESCRIPTION

perspective defines a projection transformation by indicating the field-of-view angle, fovy, in the y direction of the eye coordinate system, the aspect ratio which determines the field of view in the x direction, and the distance to the near and far clipping planes in the z direction. The aspect ratio is given as a ratio of x to y. In general, the aspect ratio given in the perspective command should match the aspect ratio of the associated viewport. For example, aspect=2.0 means the viewer sees twice as far in x as in y. If the viewport is twice as wide as it is tall, the image will be displayed without distortion. The arguments near and far are distances from the viewer to the near and far clipping planes, and these are always positive. perspective loads a matrix onto the transformation stack, overwriting whatever was there.

fovy is in 10th of degrees, as are all angles. fov must be ≥ 2 or an error will result.

SEE ALSO

ortho, window
NAME
pick - puts the system in picking mode

SPECIFICATION
C
pick(buffer, numnames)
short buffer[];
long numnames;

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

Pascal
procedure pick(buffer: Ibuffer; numnames: longint);

DESCRIPTION
pick facilitates the use of the cursor as a pointing object. When an image is
drawn in picking mode, the screen does not change. Instead, a special
viewing matrix is placed on the stack, and everything in the image that does
not intersect a small neighborhood around the lower left corner of the cursor
is discarded. The commands that do intersect the picking region are "hits"
and cause the contents of the name stack to be stored in buffer. numnames
specifies the maximum number of names to be saved by the system.

SEE ALSO
clearhitcode, endpick, endselect, gethitcode, picksize, select, pushname,
popname, loadname

NOTE
This command can be used only in immediate mode.
NAME

**picksize** - sets the dimensions of the picking window

SPECIFICATION

C

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

FORTRAN

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

Pascal

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

DESCRIPTION

**picksize** has two arguments, *deltax* and *deltay*, which define the dimensions of a rectangle, called the picking region, centered at the current cursor position, the origin of the cursor glyph. In picking mode, any objects that intersect the picking region are reported.

SEE ALSO

pick

NOTE

This command can be used only in immediate mode.
NAME

pmv - polygon move

SPECIFICATION

C

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

pmvi(x, y, z)
Icoord x, y, z;

pmvs(x, y, z)
Scoord x, y, z;

pmv2(x, y)
Coord x, y;

pmv2i(x, y)
Icoord x, y;

pmv2s(x, y)
Scoord x, y;

FORTRAN

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

subroutine pmvi(x, y, z)
integer*4 x, y, z

subroutine pmvs(x, y, z)
integer*2 x, y, z

subroutine pmv2(x, y)
real x, y

subroutine pmv2i(x, y)
integer*4 x, y

subroutine pmv2s(x, y)
integer*2 x, y

Pascal

procedure pmv(x, y, z: Coord);
procedure pmvi(x, y, z: Icoord);
procedure pmvs(x, y, z: Scoord);
procedure pmv2(x, y: Coord);
procedure pmv2i(x, y: Icoord);
procedure pmv2s(x, y: Scoord);

DESCRIPTION

pmv is the move command for filled polygons. A typical polygon is drawn with a pmv, a sequence of pdr’s, and is closed with a pclos. For example, the following sequence draws a square:

```
    pmv(0.0, 0.0, 0.0);
    pdr(1.0, 0.0, 0.0);
    pdr(1.0, 1.0, 0.0);
    pdr(0.0, 1.0, 0.0);
    pclos();
```

SEE ALSO

pclos, pdr, rpdr, rpmv

NOTE

The results are undefined if the polygon is not convex.
NAME

pnt - draws a point

SPECIFICATION

C

```c
pnt(x, y, z)
Coord x, y, z;
pnti(x, y, z)
Icoord x, y, z;
pntsi(x, y, z)
Scoord x, y, z;
pnt2(x, y)
Coord x, y;
pnt2i(x, y)
Icoord x, y;
pnt2si(x, y)
Scoord x, y;
```

FORTRAN

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

subroutine pnti(x, y, z)
integer*4 x, y, z

subroutine pntsi(x, y, z)
integer*2 x, y, z

subroutine pnt2(x, y)
real x, y

subroutine pnt2i(x, y)
integer*4 x, y

subroutine pnt2si(x, y)
integer*2 x, y
```

Pascal

```pascal
procedure pnt(x, y, z: Coord);
procedure pnti(x, y, z: Icoord);
procedure pntsi(x, y, z: Scoord);
```
procedure pnt2(x, y: Coord);
procedure pnt2i(x, y: Icoord);
procedure pnt2s(x, y: Scoord);

DESCRIPTION

pnt colors a world space point. If the point is visible in the current viewport, it will be shown as one pixel. The pixel is drawn in the current color (if in depth-cue mode, the depth-cued color is used) using the current writemask. The current graphics position is updated after the pnt command is executed.

SEE ALSO

draw, move
NAME

polarview - defines the viewer's position in polar coordinates

SPECIFICATION

C

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

FORTRAN

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

Pascal

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

DESCRIPTION

polarview defines the viewer's position in polar coordinates. The first three parameters, dist, azim, and inc, define a viewpoint. dist is the distance from the viewpoint to the world space origin. azim is the angle in the x-y plane, measured from the y axis. inc is the angle in y-z plane, measured from the z axis. The line of sight extends from the viewpoint through 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.

SEE ALSO

lookat
NAME

polf - draws a filled polygon

SPECIFICATION

C

`polf(n, parray)`
long n;
Coord parray[][3];
`polfi(n, parray)`
long n;
Icoord parray[][3];
polfs(n, parray)
long n;
Scoord parray[][3];
polf2(n, parray)
long n;
Coord parray[][2];
polf2i(n, parray)
long n;
Icoord parray[][2];
polf2s(n, parray)
long n;
Scoord parray[][2];

FORTRAN

`subroutine polf(n, parray)`
integer*4 n
real parray(3,n)
`subroutine polfi(n, parray)`
integer*4 n
integer*4 parray(3,n)
`subroutine polfs(n, parray)`
integer*4 n
integer*2 parray(3,n)
`subroutine polf2(n, parray)`
integer*4 n
real parray(2,n)
`subroutine polf2i(n, parray)`
integer*4 n
integer*4 parray(2,n)
subroutine polf2s(n, parray)
integer*4 n
integer*2 parray(2,n)

Pascal
procedure polf(n: longint; var parray: Coord3array);
procedure polfi(n: longint; var parray: Icoord3array);
procedure polfs(n: longint; var parray: Scoord3array);
procedure polf2(n: longint; var parray: Coord2array);
procedure polf2i(n: longint; var parray: Icoord2array);
procedure polf2s(n: longint; var parray: Scoord2array);

DESCRIPTION

polf fills polygonal areas using the current texture pattern, color, and writemask. It takes two arguments: an array of points and the number of points in that array. Polygons are represented as arrays of points. The first and last points are automatically connected to close a polygon. The points can be expressed as integers, shorts, or real numbers, in 2D or 3D space. Two-dimensional polygons are drawn with z=0. After the polygon is filled, the current graphics position is set to the first point in the array.

The results are undefined if the polygon is not convex.

SEE ALSO
poly, rect, rectf, pdr, pmv, rpdr, rpmv
NAME

poly - outlines a polygon

SPECIFICATION

C

poly(n, parray)
long n;
Coord parray[3];
polyi(n, parray)
long n;
Icoord parray[3];
polys(n, parray)
long n;
Scoord parray[3];
poly2(n, parray)
long n;
Coord parray[2];
poly2i(n, parray)
long n;
Icoord parray[2];
poly2s(n, parray)
long n;
Scoord parray[2];

FORTRAN

subroutine poly(n, parray)
integer*4 n
real parray(3,n)
subroutine polyi(n, parray)
integer*4 n
integer*4 parray(3,n)
subroutine polys(n, parray)
integer*4 n
integer*2 parray(3,n)
subroutine poly2(n, parray)
integer*4 n
real parray(2,n)
subroutine poly2i(n, parray)
integer*4 n

integer*4 pararray(2,n)
subroutine poly2s(n, pararray)
integer*4 n
integer*2 pararray(2,n)

Pascal

procedure poly(n: longint; var pararray: Coord3array);
procedure polyi(n: longint; var pararray: Icoord3array);
procedure polys(n: longint; var pararray: Scoord3array);
procedure poly2(n: longint; var pararray: Coord2array);
procedure poly2i(n: longint; var pararray: Icoord2array);
procedure poly2s(n: longint; var pararray: Scoord2array);

DESCRIPTION

poly takes two arguments: an array of points and the number of points in
that array. A polygon is represented as an array of points. The first and last
points are automatically connected to close the polygon. The points can be
expressed as integers, shorts, or real numbers, in 2D or 3D space. Two-
dimensional polygons are drawn with z=0. The polygon is outlined using
the current linestyle, linewidth, color, and writemask. The maximum
number of points in a polygon is 384.

SEE ALSO

polf, rect, rectf, pmv, pdr, pclos, rpmv, rpdr

NOTE

This command can be used only in immediate mode.
popattributes

NAME

popattributes - pops the attribute stack

SPECIFICATION

C

popattributes()

FORTRAN

subroutine popatt

Pascal

procedure popattributes;

DESCRIPTION

popattributes restores the most recently saved values (those pushed by pushattributes) of the global state variables:

<table>
<thead>
<tr>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>backbuffer</td>
</tr>
<tr>
<td>color</td>
</tr>
<tr>
<td>raster font</td>
</tr>
<tr>
<td>frontbuffer</td>
</tr>
<tr>
<td>linestyle</td>
</tr>
<tr>
<td>linestyle backup</td>
</tr>
<tr>
<td>writemask pattern</td>
</tr>
<tr>
<td>linewidth</td>
</tr>
<tr>
<td>lsrepeat</td>
</tr>
<tr>
<td>pattern</td>
</tr>
<tr>
<td>reset linestyle</td>
</tr>
<tr>
<td>RGB color</td>
</tr>
<tr>
<td>RGB writemask</td>
</tr>
</tbody>
</table>

An error message is printed if you attempt to pop an empty attributes stack.

SEE ALSO

backbuffer, color, defcursor, frontbuffer, linewidth, lsbackup, lsrepeat, pushattributes, resetls, RGBcolor, RGBwritemask, setlinestyle, setpattern, writemask
popmatrix

NAME

popmatrix - pops the transformation matrix stack

SPECIFICATION

C

popmatrix()

FORTRAN

subroutine popmat

Pascal

procedure popmatrix;

DESCRIPTION

popmatrix pops the transformation matrix stack. The matrix on top of the
stack before the execution of popmatrix is lost.

SEE ALSO

loadmatrix, multmatrix, pushmatrix
NAME
  popname - pops a name off the name stack

SPECIFICATION
  C       popname()

  FORTRAN subroutine popnam

  Pascal  procedure popname;

DESCRIPTION
  popname removes the top name from the name stack (used in picking and selecting modes).

SEE ALSO
  loadname, pushname, pick, select
NAME

popviewport - restores the viewport, screenmask, and setdepth parameters

SPECIFICATION

C       popviewport()

FORTRAN subroutine popvie

Pascal  procedure popviewport;

DESCRIPTION

The popviewport command pops the stack of viewports and screenmasks. The viewports on top of the stack are lost. In addition, popviewport restores the setdepth parameters.

SEE ALSO

getscrmask, pushviewport, setdepth, viewport
NAME

cprefposition - specifies the preferred location and size of a graphics port

SPECIFICATION

C

cprefposition(x1, x2, y1, y2)
long x1, x2, y1, y2;

FORTRAN

subroutine prefpo(x1, x2, y1, y2)
integer*4 x1, x2, y1, y2

Pascal

procedure prefposition(x1, x2, y1, y2: long);

DESCRIPTION

cprefposition specifies the preferred location and size of a graphics port. The location is specified in screen coordinates. The window manager will not allow the graphics port to be relocated or resized.

cprefposition is called at the beginning of a graphics program that will be run with the window manager. If getport is not called or if the system is not running the window manager, cprefposition will be ignored.

SEE ALSO

getport

NOTE

This command can be used only in immediate mode.
NAME

`prefsize` - specifies the preferred size of a graphics port in the window manager

SPECIFICATION

C

```c
prefsize(x, y)
long x, y;
```  

FORTRAN

```fortran
subroutine prefsi(x, y)
integer*4 x, y
```  

Pascal

```pascal
procedure prefsize(x, y: long);
```  

DESCRIPTION

`prefsize` specifies that the size of a graphics port should be $x$ pixels by $y$ pixels. It is called at the beginning of a graphics program that will be run with the window manager. The window manager will not allow the graphics port to be resized. `prefsize` will be ignored if `getport` is not called or if the system is not running the window manager.

SEE ALSO

`getport`

NOTE

This command can be used only in immediate mode.
pushattributes

NAME

pushattributes - pushes attributes on a stack

SPECIFICATION

C

pushattributes()

FORTRAN

subroutine pushat

Pascal

procedure pushattributes;

DESCRIPTION

pushattributes saves the global state. A stack of attributes is maintained, and pushattributes puts copies of global state variables on the stack. The global state variables that will be saved are:

<table>
<thead>
<tr>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>backbuffer</td>
</tr>
<tr>
<td>color</td>
</tr>
<tr>
<td>raster font</td>
</tr>
<tr>
<td>frontbuffer</td>
</tr>
<tr>
<td>linestyle</td>
</tr>
<tr>
<td>linestyle backup</td>
</tr>
<tr>
<td>writemask</td>
</tr>
<tr>
<td>linewidth</td>
</tr>
<tr>
<td>lsrepeat</td>
</tr>
<tr>
<td>reset linestyle</td>
</tr>
<tr>
<td>RGB color</td>
</tr>
<tr>
<td>RGB writemask</td>
</tr>
<tr>
<td>pattern</td>
</tr>
</tbody>
</table>

The attribute stack is 12 levels deep. pushattributes is ignored if the stack is full.

SEE ALSO

backbuffer, color, frontbuffer, linewidth, lsbackup, lsrepeat, popattributes, resetls, RGBcolor, RGBwritemask, setlinestyle, settexture, writemask
NAME

pushmatrix - pushes down the transformation matrix stack

SPECIFICATION

C       pushmatrix()

FORTRAN subroutine pushma

Pascal  procedure pushmatrix;

DESCRIPTION

pushmatrix pushes down the transformation matrix stack, duplicating the current matrix; i.e., if the stack contains one matrix, after a pushmatrix command it will contain two copies of. The top copy can be modified. The stack is eight levels deep in hardware, and extends to 32 levels in software.

SEE ALSO

loadmatrix, multmatrix, popmatrix
NAME

pushname - pushes a new name on the name stack

SPECIFICATION

C
pushname(name)
short name;

FORTRAN
subroutine pushna(name)
integer*4 name

Pascal
procedure pushname(name: Short);

DESCRIPTION

pushname pushes the name stack down one level, and puts a new 16-bit name on top. The contents of the name stack are stored in a buffer for each "hit" in picking and selecting modes.

SEE ALSO

popname, loadname, pick, select
NAME

pushviewport - save the current viewport, screenmask, and setdepth parameters

SPECIFICATION

C       pushviewport()

FORTRAN subroutine pushvi

Pascal  procedure pushviewport;

DESCRIPTION

The current viewport is the top element in a stack of viewports. The pushviewport command duplicates the current viewport and pushes it on the stack. After pushviewport, there are two copies of the current viewport in the stack; the top one can be changed without losing the old one. In addition, it saves the screenmask and the parameters specified by the setdepth command.

SEE ALSO

getscrmask, popviewport, setdepth, viewport
NAME

qdevice - queues a device (keyboard, button, or valuator)

SPECIFICATION

C
qdevice(v)
Device v;

FORTRAN
subroutine qdevic(v)
integer*4 v

Pascal
procedure qdevice(v: Device);

DESCRIPTION

qdevice causes changes in the state of the named device to be entered in the event queue. The device can be a keyboard key, a button, or a valuator.

SEE ALSO

unqdevice, noise, tie

NOTE

This command can be used only in immediate mode.
NAME
qenter - creates an event queue entry

SPECIFICATION

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

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

Pascal
procedure qenter(qtype, val: Short);

DESCRIPTION
qenter takes two 16-bit integers, qtype and value, and enters them into the event queue. There is no way to distinguish user-defined and system-defined entries unless disjoint sets of types are used. See Appendix A for a list of system-defined types.

SEE ALSO
qreset, qtest, qread

NOTE
This command can be used only in immediate mode.
qread

NAME

qread - reads the first entry in the event queue

SPECIFICATION

C
long qread(data)
short *data;

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

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

DESCRIPTION

qread waits until there is an entry in the queue. It returns the device
number of a queue entry, writes the data part of the entry into data, and
removes the entry from the queue.

SEE ALSO

qtest, qreset

NOTE

This command can be used only in immediate mode.
NAME

qreset - empties the event queue

SPECIFICATION

C

qreset()

FORTRAN

subroutine qreset

Pascal

procedure qreset;

DESCRIPTION

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

SEE ALSO

qenter, qread, qtest

NOTE

This command can be used only in immediate mode.
NAME
qtest - checks the contents of the event queue

SPECIFICATION
C
long qtest()

FORTRAN
integer*4 function qtest()

Pascal
function qtest: long;

DESCRIPTION
qtest returns zero if the queue is empty. Otherwise, the device number of
the first entry is returned, and the queue remains unchanged.

SEE ALSO
qenter, qread, qreset

NOTE
This command can be used only in immediate mode.
NAME

rdr - relative draw

SPECIFICATION

C

rdr(dx, dy, dz)
Coord dx, dy, dz;
rdr1(dx, dy, dz)
Icoord dx, dy, dz;
rdr2(dx, dy, dz)
Scoord dx, dy, dz;
rdr2s(dx, dy)
Scoord dx, dy;

FORTRAN

subroutine rdr(dx, dy, dz)
real dx, dy, dz
subroutine rdr1(dx, dy, dz)
integer*4 dx, dy, dz
subroutine rdr2(dx, dy)
real dx, dy
subroutine rdr2s(dx, dy)
integer*2 dx, dy

Pascal

procedure rdr(dx, dy, dz: Coord);
procedure rdr1(dx, dy, dz: Icoord);
procedure rdr2s(dx, dy, dz: Scoord);
procedure rdr2(dx, dy: Coord);
procedure rdr2i(dx, dy: Icoord);
procedure rdr2s(dx, dy: Scoord);

DESCRIPTION

rdr is the relative version of the draw command. It connects the current graphics position and the point at the specified distance away with a line segment using the current linestyle, linewidth, color (if in depth cue mode, the depth cued color is used), and writemask. The current graphics position is updated to the new point. Commands that invalidate the current graphics position should not be placed within sequences of relative moves and draws.

SEE ALSO

draw, move, rmv
NAME

readpixels - returns values of specific pixels

SPECIFICATION

C

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

FORTRAN

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

Pascal

function readpixels(n: Short; var colors: Colorarray): longint;

DESCRIPTION

readpixels attempts to read up to n pixel values from the bitplanes. They are read into the array colors starting from the current character position along a single scan line (constant y) in the direction of increasing x. readpixels returns the number of pixels actually read, which is only guaranteed to be the number requested if the starting point is at least that many away from the edge of the current viewport. The values of pixels read outside the current viewport are undefined. The current character position is updated to be one pixel to the right of the last one read, or is undefined if the new position is outside the viewport.

In double buffer mode only the back buffer is read. Use readRGB to read pixels in RGB mode.

SEE ALSO

readRGB, writepixels

NOTE

This command can be used only in immediate mode.
NAME

readRGB - returns values of specific pixels

SPECIFICATION

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

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

Pascal
function readRGB(n: Short; var red, green, blue:
RGBArray): longint;

DESCRIPTION

readRGB attempts to read up to n pixel values from the bitplanes. 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 actually read, which is only
guaranteed to be the number requested if the starting point is at least that
many away from the edge of the current viewport. The values of pixels read
outside the current viewport are undefined. The current character position
is updated to be one pixel to the right of the last one read, or is undefined if
the new position is outside the viewport.

SEE ALSO

readpixels, writeRGB

NOTE

This command can be used only in immediate mode.
NAME

rect - outlines a rectangular region

SPECIFICATION

C
rect(x1, y1, x2, y2)
Coord x1, y1, x2, y2;
recti(x1, y1, x2, y2)
Icoord x1, y1, x2, y2;

FORTRAN
subroutine rect(x1, y1, x2, y2)
real x1, y1, x2, y2
subroutine recti(x1, y1, x2, y2)
integer*4 x1, y1, x2, y2

Pascal
procedure rect(x1, y1, x2, y2: Coord);
procedure recti(x1, y1, x2, y2: Icoord);

DESCRIPTION

rect draws a rectangle using the current linestyle, linewidth, color, and writemask. The sides of the rectangle are parallel to the x and y coordinate system. Since a rectangle is a two-dimensional shape, rect takes only 2D arguments, and sets the z coordinate to zero. The points (x1, y1) and (x2, y2) are the opposite corners of the rectangle.

SEE ALSO

poly, rectf, rpdr, rpmv
NAME

rectcopy - copies a rectangle of pixels on the IRIS screen

SPECIFICATION

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

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

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

DESCRIPTION

rectcopy copies a rectangular array of pixels defined in screen coordinates to another position on the screen whose lower-left corner is defined by the point (newx, newy). The drawing of the copied region is masked by the current viewport and screenmask.

NOTE

This command can be used only in immediate mode.
NAME

rectf - fills a rectangular area

SPECIFICATION

C
rectf(x1, y1, x2, y2)
Coord x1, y1, x2, y2;
rectfi(x1, y1, x2, y2)
Icoord x1, y1, x2, y2;

FORTRAN
subroutine rectf(x1, y1, x2, y2)
real x1, y1, x2, y2
subroutine rectfi(x1, y1, x2, y2)
integer*4 x1, y1, x2, y2

Pascal
procedure rectf(x1, y1, x2, y2: Coord);
procedure rectfi(x1, y1, x2, y2: Icoord);

DESCRIPTION

rectf produces a filled rectangular region, using the current texture pattern, color, and writemask. The sides of the rectangle are parallel to the x and y axes of the object coordinate system. Since a rectangle is a two-dimensional shape, rectf takes only 2D arguments, and sets the z coordinate to zero. The points (x1, y1) and (x2, y2) are the opposite corners of the rectangle. The current graphics position is set to (x1, y1) after the region is drawn.

SEE ALSO

polf, rect, rdr, rmv
NAME
resetls - controls the continuity of linestyles

SPECIFICATION

C
resetls(b)
Boolean b;

FORTRAN
subroutine resetl(b)
logical b

Pascal
procedure resetls(b: Boolean);

DESCRIPTION
resetls affects the re-initialization of the linestyle pattern between segments. It takes one boolean argument. TRUE, the default, means that stippling of each line will start from the beginning of the linestyle pattern. FALSE turns the mode off: the linestyle will not be reset between segments, and the stippling of one segment will continue from where it left off at the end of the previous segment. Calls to resetls have the side effect of initializing the linestyle, no matter what the argument, and of invalidating the current graphics position.

resetls is used most often when approximating circles, arcs, and curves with many short lines. If the linestyle is not reset between segments, the pattern of the curve appears smooth and continuous. The linewidth should not be set to 2 unless resetls is TRUE.

SEE ALSO
deflinestyle, getresetls, lsbackup, setlinestyle
NAME

reshapeviewport - sets the viewport to the current dimensions of the graphics port in the window manager

SPECIFICATION

C

reshapeviewport()

FORTRAN

subroutine reshap

Pascal

procedure reshapeviewport;

DESCRIPTION

reshapeviewport sets the viewport to the current dimensions of the graphics port.

reshapeviewport is equivalent to:

\{
    int xmin, xmax, ymin, ymax;
    getorigin(&xmin, &ymin);
    getsize(&xmax, &ymax);
    xmax += xmin -1;
    ymax += ymin -1;
    viewport(xmin, xmax, ymin, ymax);
\}

SEE ALSO

getorigin, getsize, viewport

NOTE

This command can be used only in immediate mode.
NAME

RGBcolor - sets the current color in RGB mode

SPECIFICATION

C

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

FORTRAN

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

Pascal

procedure RGBcolor(red, green, blue: RGBvalue);

DESCRIPTION

In RGB mode, the current color is set with the RGBcolor command. Its arguments are three eight-bit values, one each for red, green, and blue. These numbers are written directly into the bitplanes whenever a pixel is drawn; they control the intensity of red, green, and blue displayed on the screen. RGBcolor is ignored in single or double buffer mode.

SEE ALSO

color, gRGBcolor, RGBwritemask
NAME

RGBcursor - sets the characteristics of the cursor in RGB mode

SPECIFICATION

C

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

FORTRAN

subroutine RGBcur(index, red, green, blue, redm,
greenm, bluem)
integer*4 index, red, green, blue, redm, greenm, bluem

Pascal

procedure RGBcursor(index: Short;
red, green, blue, redm, greenm, bluem: RGBvalue);

DESCRIPTION

RGBcursor allows selection of a cursor glyph from a table of 16×16 bit patterns already defined by the user. The 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 can be used only in RGB mode.

SEE ALSO

RGBmode, RGBwritemask, setcursor, defcursor

NOTE

This command can be used only in immediate mode.
NAME

RGBmode - sets a display mode that bypasses the color map

SPECIFICATION

C       RGBmode()

FORTRAN subroutine RGBmod

Pascal  procedure RGBmode;

DESCRIPTION

There are three display modes: single buffer, double buffer, and RGB. In RGB mode, all the bitplanes are simultaneously written and displayed. Eight-bit values of red, green, and blue are written into the bitplanes, and directly control the monitor's intensity. RGB mode is useful only when the system has 24 bitplanes: the first sixteen are shared equally between red and green; the last eight define the blue component. A twelve-plane system provides one-quarter the maximum intensity in red and green, and no blue at all. RGBmode does not take effect until gconfig is called.

SEE ALSO

doublebuffer, gconfig, getdisplaymode, singlebuffer

NOTE

This command can be used only in immediate mode.
NAME

RGBwritemask - grants write access to a subset of the available bitplanes

SPECIFICATION

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

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

Pascal
procedure RGBwritemask(red, green, blue: RGBvalue);

DESCRIPTION

RGBwritemask shields bitplanes reserved for special uses from ordinary
drawing commands in RGB mode. The three arguments are masks for each
of the three sets of eight planes. Wherever there are ones in the writemask,
the corresponding bits in the RGB color will be written into the bitplanes.
Zeros in the writemask mark bitplanes as read-only. These planes will not
be changed, regardless of the bits in the RGB color.

SEE ALSO

gRGBmask, RGBcolor, writemask
NAME

ringbell - rings the keyboard bell

SPECIFICATION

C      ringbell()

FORTRAN subroutine ringbe

Pascal procedure ringbell;

DESCRIPTION

ringbell rings the keyboard bell.

SEE ALSO

clockoff, clockon, lampoff, lampon, setbell

NOTE

This command can be used only in immediate mode.
NAME

rmv - relative move

SPECIFICATION

C

rmv(dx, dy, dz)
Coord dx, dy, dz;
rmvi(dx, dy, dz)
Icoord dx, dy, dz;
rmvs(dx, dy, dz)
Scoord dx, dy, dz;
rmv2(dx, dy)
Coord dx, dy;
rmv2i(dx, dy)
Icoord dx, dy;
rmv2s(dx, dy)
Scoord dx, dy;

FORTRAN

subroutine rmv(dx, dy, dz)
real dx, dy, dz
subroutine rmvi(dx, dy, dz)
integer*4 dx, dy, dz
subroutine rmvs(dx, dy, dz)
integer*2 dx, dy, dz
subroutine rmv2(dx, dy)
real dx, dy
subroutine rmv2i(dx, dy)
integer*4 dx, dy
subroutine rmv2s(dx, dy)
integer*2 dx, dy

Pascal

procedure rmv(dx, dy, dz: Coord);
procedure rmvi(dx, dy, dz: Icoord);
procedure rmvs(dx, dy, dz: Scoord);
procedure rmv2(dx, dy: Coord);
procedure rmv2i(dx, dy: Icoord);
procedure rmv2s(dx, dy: Scoord);

DESCRIPTION

rmv is the relative version of the move command. It moves (without draw-
ing) the graphics position the specified amount relative to its current value. The command has six forms: 3D floating point, 3D integer, 2D floating point, 2D integer, 3D short integer, and 2D short integer. rmv2(x, y) is equivalent to rmv(x, y, 0.0).

SEE ALSO

draw, move, rdr
NAME

rotate - rotates graphical primitives

SPECIFICATION

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

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

Pascal
procedure rotate(a: Angle; axis: char);

DESCRIPTION

rotate specifies an angle and an axis of rotation. The angle is given in tenths
of degrees according to the right-hand rule. The axis of rotation is defined
by a character, either 'x', 'y', or 'z' (the character can be upper or lower
case). rotate is a modeling command; thus, it changes the current transfor-
mation matrix. All objects drawn after the rotate command is executed will
be rotated. Use pushmatrix and popmatrix to preserve and restore an unro-
tated world space.

SEE ALSO

popmatrix, pushmatrix, scale, translate
NAME

rpdr - relative polygon draw

SPECIFICATION

C

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

rpdrri(dx, dy, dz)
lcoord dx, dy, dz;

rpdr2s(dx, dy, dz)
Scoord dx, dy, dz;

rpdr2(dx, dy)
Coord dx, dy;

rpdr2i(dx, dy)
lcoord dx, dy;

rpdr2s(dx, dy)
Scoord dx, dy;

FORTRAN

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

subroutine rpdrri(dx, dy, dz)
integer*4 dx, dy, dz

subroutine rpdr2s(dx, dy, dz)
integer*2 dx, dy, dz

subroutine rpdr2(dx, dy)
real dx, dy

subroutine rpdr2i(dx, dy)
integer*4 dx, dy

subroutine rpdr2s(dx, dy)
integer*2 dx, dy

Pascal

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

procedure rpdrri(dx, dy, dz: lcoord);

procedure rpdr2s(dx, dy, dz: Scoord);
procedure rpdr2(dx, dy: Coord);
procedure rpdr2i(dx, dy: Icoord);
procedure rpdr2s(dx, dy: Scoord);

DESCRIPTION

rpdr is the relative version of the pdr command. It creates an edge of a filled polygon between the current graphics position and the point at the specified distance away. The current graphics position is updated to the new point. The results are undefined if the polygon is not convex.

SEE ALSO

pclos, pdr, pmv, rpv
NAME

rpmv - relative polygon move

SPECIFICATION

C

rpmv(dx, dy, dz)
Coord dx, dy, dz;
rpmvi(dx, dy, dz)
Icoord dx, dy, dz;
rpmvs(dx, dy, dz)
Scoord dx, dy, dz;
rpmv2(dx, dy)
Coord dx, dy;
rpmv2i(dx, dy)
Icoord dx, dy;
rpmv2s(dx, dy)
Scoord dx, dy;

FORTRAN

subroutine rpmv(dx, dy, dz)
real dx, dy, dz
subroutine rpmvi(dx, dy, dz)
integer*4 dx, dy, dz
subroutine rpmvs(dx, dy, dz)
integer*2 dx, dy, dz
subroutine rpmv2(dx, dy)
real dx, dy
subroutine rpmv2i(dx, dy)
integer*4 dx, dy
subroutine rpmv2s(dx, dy)
integer*2 dx, dy

Pascal

procedure rpmv(dx, dy, dz: Coord);
procedure rpmvi(dx, dy, dz: Icoord);
procedure rpmvs(dx, dy, dz: Scoord);

Version 2.1
procedure rpmv2(dx, dy: Coord);
procedure rpmv2i(dx, dy: Icoord);
procedure rpmv2s(dx, dy: Scoord);

DESCRIPTION

rpmv is the relative version of the pmv command. It starts a filled polygon at the point at the specified distance from the current graphics position. The current graphics position is updated to the new point. The results are undefined if the polygon is not convex.

SEE ALSO

pclos, pdr, pmv, rpdr
NAME

scale - scales and mirrors objects

SPECIFICATION

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

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

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

DESCRIPTION

scale shrinks, expands, and mirrors objects. Its three arguments specify scaling in each of the three coordinate directions. Values with magnitude of more than 1 expand the object; values with magnitude less than 1 shrink it. Negative values cause mirroring. scale is a modeling command; it changes the current transformation matrix. All objects drawn after the scale command is executed will be affected. Use pushmatrix and popmatrix to limit the scope of the scale command.

SEE ALSO

popmatrix, pushmatrix, rotate, translate
NAME

screenspace - puts a program in screen space under the window manager

SPECIFICATION

C
screenspace()

FORTRAN
subroutine screen

Pascal
procedure screenspace;

DESCRIPTION

screenspace puts the specified program in screen space. Graphics positions are expressed in absolute screen coordinates. This allows pixels and locations outside a program's graphics port to be read.

screenspace is equivalent to:

{  
  int xmin, ymin;
  getorigin(&xmin, &ymin);
  viewport(-xmin, 1023-xmin, -ymin, 767-ymin);
  ortho2(-0.5, 1023.5, -0.5, 767.5);
}

SEE ALSO

getorigin, viewport, ortho2

NOTE

This command can be used only in immediate mode.
NAME


crmask - defines a clipping mask for the screen

SPECIFICATION

C

crmask(left, right, bottom, top)
Screencord left, right, bottom, top;

FORTRAN

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

Pascal

procedure crmask(left, right, bottom, top: Screencord);

DESCRIPTION


crmask makes it possible to do fine character clipping. A call to viewport
sets both the viewport and the screenmask to the same area, which is
defined by the left, right, bottom, and top boundaries. A call to crmask sets
only the screenmask, which should be placed entirely within the viewport.
Strings that begin outside the viewport will be clipped out; this is "gross
clipping". Strings which begin inside the viewport but outside the screen-
mask will be clipped to the pixel boundaries of the screenmask. This is
called "fine clipping". All drawing commands are also clipped to this
viewport, but it is only useful for characters; gross clipping is sufficient for
all other primitives.

SEE ALSO

getscrmask, viewport
**NAME**

*select* - puts the IRIS in selecting mode

**SPECIFICATION**

**C**

```c
select(buffer, numnames)
short buffer[0];
long numnames;
```

**FORTRAN**

```fortran
subroutine select(buffer, numnam)
integer*2 buffer(0)
integer*4 numnam
```

**Pascal**

```pascal
procedure select(buffer: lbuffer; numnames: longint);
```

**DESCRIPTION**

*select* turns on the selecting mode. The viewing matrix in use when *select* is executed defines the selecting region. Alternatively, however, a viewing matrix can be constructed after selecting mode has begun. *select* and *pick* are identical except for the ability to create a viewing matrix in selection mode. *numnames* specifies the maximum number of names to be saved by the system. Names are 16-bit numbers, and are loaded onto the name stack by the user. Each drawing command that intersects the selecting region causes the contents of the name stack to be stored in *buffer*.

**SEE ALSO**

endpick, endselect, pick, picksize, initnames pushname, popname, loadname

**NOTE**

This command can be used only in immediate mode.
NAME

`setbell` - sets the duration of the keyboard bell

SPECIFICATION

C

```c
setbell(mode)
char mode;
```

FORTRAN

```fortran
subroutine setbel(mode)
integer*4 mode
```

Pascal

```pascal
procedure setbell(mode: Byte);
```

DESCRIPTION

`setbell` sets the duration of the keyboard bell.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>off</td>
</tr>
<tr>
<td>1</td>
<td>short beep</td>
</tr>
<tr>
<td>2</td>
<td>long beep</td>
</tr>
</tbody>
</table>

SEE ALSO

`clkoff, clkon, lampoff, lampon, ringbell`

NOTE

This command can be used only in immediate mode.
NAME

setcursor - sets the cursor characteristics

SPECIFICATION

C

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

FORTRAN

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

Pascal

procedure setcursor(index: Short; color, wtm: Colorindex);

DESCRIPTION

setcursor selects a cursor glyph from among those the user has defined with defcursor. The first argument, index, picks a glyph from the definition table. color and wtm select a color and writemask for the cursor. The default cursor is 0; it is displayed with the color 1 drawn in the first available bitplane, and is automatically updated and displayed on each vertical retrace.

SEE ALSO

attachcursor, defcursor, getcursor, RGBcursor, curorigin

NOTE

This command can be used only in immediate mode.
NAME

**setdblights** - sets the lights on the dial and button box

SPECIFICATION

C

```c
setdblights(mask)
long mask;
```

FORTRAN

```fortran
subroutine setdbl(mask)
integer*4 mask
```

Pascal

```pascal
procedure setdblights(mask: long);
```

DESCRIPTION

**setdblights** turns on some combination of the lights on the dial and button box. Each bit in the mask corresponds to a light. To turn on lights 4, 7 and 22 and all the others off, the mask should be set to \((1 << 4) | (1 << 7) | (1 << 22)\) = 400090.

NOTE

This command can be used only in immediate mode.
NAME

setdepth - sets up a 3D viewport

SPECIFICATION

C

setdepth(near, far)
Screencoord near, far;

FORTRAN

subroutine setdep(near, far)
integer*4 near, far

Pascal

procedure setdepth(near, far: Screencoord);

DESCRIPTION

viewport specifies a mapping from the left, right, bottom, and top clipping planes in world coordinates to screen coordinate values. setdepth completes this mapping for homogeneous world coordinates. The two arguments map the near and far clipping planes to the desired screen coordinate values. The default is setdepth(0, 1023).

SEE ALSO

pushviewport, popviewport, feedback, endfeedback
NAME

setlinestyle - selects a linestyle

SPECIFICATION

C

setlinestyle(index)
short index;

FORTRAN

subroutine setlin(index)
integer*4 index

Pascal

procedure setlinestyle(index: Short);

DESCRIPTION

setlinestyle chooses a linestyle pattern. Its argument is an index into the linestyle table built by calls to deflinestyle. There is always a current linestyle; it is used for drawing lines and curves and for outlining rectangles, polygons, circles, and arcs. The default linestyle, 0, is a solid line and cannot be redefined.

SEE ALSO

deflinestyle, getlstyle, linewidth, lsbackup, resetls
NAME

setmap - chooses one of the sixteen small color maps

SPECIFICATION

C

    setmap(mapnum)
    short mapnum;

FORTRAN

    subroutine setmap(mapnum)
    integer*4 mapnum

Pascal

    procedure setmap(mapnum: Short);

DESCRIPTION

setmap chooses one of the sixteen small maps, numbered 0 through 15. setmap is ignored in singlemap mode.

SEE ALSO

getmap, multimap, onemap

NOTE

This command can be used only in immediate mode.
NAME

setmonitor - sets the monitor type

SPECIFICATION

C

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

FORTRAN

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

Pascal

```pascal
procedure setmonitor(type: Short)
```

DESCRIPTION

setmonitor sets the monitor to 30 Hz interlaced, 60 Hz non-interlaced, or NTSC, depending on whether type is HZ30, HZ60, or NTSC, respectively. Those constants are defined in the file `get.h`.

<table>
<thead>
<tr>
<th>TYPE</th>
<th>Monitor Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>HZ30</td>
<td>30 Hz interlaced</td>
</tr>
<tr>
<td>HZ60</td>
<td>60 Hz non-interlaced</td>
</tr>
<tr>
<td>NTSC</td>
<td>NTSC</td>
</tr>
</tbody>
</table>

SEE ALSO

getmonitor

NOTE

This command can be used only in immediate mode.
NAME

setpattern - selects a texture pattern for filling polygons, rectangles, and curves

SPECIFICATION

C

    setpattern(index)
    short index;

FORTRAN

    subroutine setpat(index)
    integer*4 index

Pascal

    procedure setpattern(index: Short);

DESCRIPTION

setpattern selects a texture pattern from among those the user has defined with defpattern. The default pattern is pattern 0, which is solid. If an undefined pattern is specified, the default pattern is selected.

SEE ALSO

color, defpattern, getpattern, writemask
NAME

**setshade** - sets the current polygon shade

SPECIFICATION

C

```c
setshade(shade)
Colorindex shade;
```

FORTRAN

```fortran
subroutine setsha(shade)
integer*4 shade
```

Pascal

```pascal
procedure setshade(shade: Colorindex);
```

DESCRIPTION

**setshade** sets the current shade value. This value is a color index which is associated with the vertices specified immediately following the **setshade** command. The **setshade** values are used to shade polygons using **spclos**.

SEE ALSO

spclos, splf
NAME

`setvaluator` - assigns an initial value to a valuator

SPECIFICATION

C

```c
setvaluator(v, init, min, max)
Device v;
short init, min, max;
```

FORTRAN

```fortran
subroutine setval(v, init, min, max)
integer*4 v, init, min, max
```

Pascal

```pascal
procedure setvaluator(v: Device; init, min, max: Short);
```

DESCRIPTION

`setvaluator` assigns an initial value `init` to a valuator. Some devices, like tablets, report values fixed to a grid. In this case, the device defines an initial position and `init` will be ignored. `min` and `max` are lower and upper bounds for the values the device can assume.

SEE ALSO

`getvaluator`

NOTE

This command can be used only in immediate mode.
NAME

**shaderange** - sets range of color indices

SPECIFICATION

**C**

```
shaderange(lowindex, highindex, z1, z2)
Colorindex lowindex, highindex;
Screencoord z1, z2;
```

**FORTRAN**

```
subroutine shader(lowindex, highindex, z1, z2)
integer*4 lowindex, highindex, z1, z2
```

**Pascal**

```
procedure shaderange(lowindex, highindex: Colorindex;
                    z1, z2: Screencoord);
```

DESCRIPTION

`shaderange` sets the range of color indices used in drawing depthcued lines and points. The range of values specified by `z1, z2` is mapped linearly into the range of color indices used to draw lines and points.

SEE ALSO

depthcue
NAME

singlebuffer - writes and displays all the bitplanes

SPECIFICATION

C

singlebuffer()

FORTRAN

subroutine single

Pascal

procedure singlebuffer;

DESCRIPTION

The IRIS has three display modes: single buffer, double buffer, and RGB. In
single buffer mode, all the available bitplanes are simultaneously updated
and displayed. Incomplete or changing pictures appear on the screen.
singlebuffer invokes this buffer mode. It does not take effect until gconfig
is called.

SEE ALSO

doublebuffer, gconfig, getdisplaymode, gsnc, RGBmode

NOTE

This command can be used only in immediate mode.
spclos

NAME

spclos - draws currently open polygon

SPECIFICATION

C       spclos()

FORTRAN subroutine spclos

Pascal  procedure spclos();

DESCRIPTION

spclos, like pclos causes the currently "open" polygon to be drawn. spclos
draws a Gouraud-shaded polygon using intensities specified by the setshade
command.

SEE ALSO

setshade, splf, pclos, pdr, pmv
NAME

splf - draws a shaded filled polygon

SPECIFICATION

C

splf(n, pararray, iararray)
long n;
Coord pararray[3];
Colorindex iararray[];
splfi(n, pararray, iararray)
long n;
Icoord pararray[3];
Colorindex iararray[];
splfs(n, pararray, iararray)
long n;
Scoord pararray[3];
Colorindex iararray[];
splf2(n, pararray, iararray)
long n;
Coord pararray[2];
Colorindex iararray[];
splf2i(n, pararray, iararray)
long n;
Icoord pararray[2];
Colorindex iararray[];
splf2s(n, pararray, iararray)
long n;
Scoord pararray[2];
Colorindex iararray[];

FORTRAN

subroutine splf(n, pararray, iararray)
icient*4 n
real pararray(3,n)
icent*2 iararray(n)

subroutine splfi(n, pararray, iararray)
icent*4 n
icent*4 pararray(3,n)
icent*2 iararray(n)
subroutine splfs(n, parray, iarray)
integer*4 n
integer*2 parray(3,n)
integer*2 iarray(n)

subroutine splf2(n, parray, iarray)
integer*4 n
real parray(2,n)
integer*2 iarray(n)

subroutine splf2i(n, parray, iarray)
integer*4 n
integer*4 parray(2,n)
integer*2 iarray(n)

subroutine splf2s(n, parray, iarray)
integer*4 n
integer*2 parray(2,n)
integer*2 iarray(n)

Pascal

procedure splf(n: longint; var parray: Coord3array;
var iarray: Colorarray);

procedure splf2(n: longint; var parray: Icoord3array;
var iarray: Colorarray);

procedure splf2i(n: longint; var parray: Icoordarray;
var iarray: Colorarray);

procedure splf2s(n: longint; var parray: Scoordarray;
var iarray: Colorarray);

procedure splf2i(n: longint; var parray: Icoord2array;
var iarray: Colorarray);

procedure splf2s(n: longint; var parray: Scoord2array;
var iarray: Colorarray);

DESCRIPTION

**splf** draws Gouraud-shaded polygonal areas using the current texture pattern and writemask. It takes three arguments: an array of points, an array of the intensities at these points, and the number of points in each array. Polygons are represented as arrays of points. The first and last points are automatically connected to close a polygon. The points can be expressed as integers, shorts, or real numbers, in 2D or 3D space. Two-dimensional polygons are drawn with $z = 0$. After the polygon is drawn, the current graphics position is set to the first point in the array. The results are undefined if the polygon is not convex.
SEE ALSO

poly, rect, rectf, pdr, pmv, rpdr, rpmv
NAME

stepunit - specifies that a graphics port should change size in discrete steps

SPECIFICATION

C

    stepunit(xunit, yunit)
    long xunit, yunit;

FORTRAN

    subroutine stepun(xunit, yunit)
    integer*4 xunit, yunit

Pascal

    procedure stepunit(xunit, yunit: long);

DESCRIPTION

stepunit specifies that the size of a graphics port will change in discrete steps of xunit in the x direction and yunit in the y direction. It is called at the beginning of a graphics program that will be run with the window manager and takes effect when getport is called. It is used to resize graphics ports in units of a standard size (in pixels). When stepunit is called, the dimensions of the graphics port will be:

    width = xunit*n
    height = yunit*m

If getport is not called, or if the system is not running the window manager, stepunit will be ignored.

SEE ALSO

getport, fudge

NOTE

This command can be used only in immediate mode.
NAME

strwidth - returns the width of the specified text string

SPECIFICATION

C

long strwidth(str)
String str;

FORTRAN

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

Pascal

function strwidth(str: pstring): longint;

DESCRIPTION

strwidth returns the width, in pixels, of a text string, using the character spacing parameters of the currently selected raster font. strwidth is useful only when there is a simple mapping from screen to world space. The user must do the mapping.

Undefined characters have zero width.

In FORTRAN, there are two arguments to strwid : str is the name of the string and length is the number of characters in that string.

SEE ALSO

gheight, getlwidth, mapw, mapw2

NOTE

This command can be used only in immediate mode.
NAME

swapbuffers - swaps the front and back buffers in double buffer mode

SPECIFICATION

C

swapbuffers()

FORTRAN

subroutine swapbu

Pascal

procedure swapbuffers;

DESCRIPTION

swapbuffers swaps the front and back buffers in double buffer mode during the next vertical retrace period. After an image is drawn in the back buffer, a swapbuffers command displays it. swapbuffers is ignored in single buffer or RGB mode.

SEE ALSO

backbuffer, doublebuffer, frontbuffer, swapinterval, gsync
NAME

swapinterval - defines a minimum time between buffer swaps

SPECIFICATION

C

    swapinterval(i)
    short i;

FORTRAN

    subroutine swapin(i)
    integer*4 i

Pascal

    procedure swapinterval(i: Short);

DESCRIPTION

swapinterval establishes a minimum time between buffer swaps. If the user
specifies a swap interval of 5, for example, the screen will be refreshed at
least five times between execution of successive swapbuffers commands.
This command provides a way to change frames at a steady rate if a new
image can be created within one swap interval. The default interval is 1.
swapinterval is valid only in double buffer mode. It is ignored in single
buffer or RGB mode.

SEE ALSO

doublebuffer, swapbuffers, gsync

NOTE

This command can be used only in immediate mode.
NAME
textcolor - sets the color of text drawn in the textport

SPECIFICATION

C           textcolor(tcolor)
            Colorindex tcolor;

FORTRAN     subroutine textco(tcolor)
            integer*4 tcolor

Pascal      procedure textcolor(tcolor: Colorindex);

DESCRIPTION
textcolor sets the color of the text drawn in the textport. The color of text
drawn using charstr is determined by color.

SEE ALSO
color, pagecolor, pagewritemask, textport, textwritemask, tpoft, tpon

NOTE
This command can be used only in immediate mode.
NAME
   textinit - initializes the textport

SPECIFICATION
   C       textinit()
   FORTRAN subroutine textin
   Pascal  procedure textinit;

DESCRIPTION
   textinit initializes the textport to default size, location, textcolor, pagecolor, and textwritemask.

NOTE
   This command can only be used in immediate mode.

SEE ALSO
   textport, textcolor, pagecolor, textwritemask

NOTE
   This command can be used only in immediate mode.
NAME
textport - allocates an area of the screen for the textport

SPECIFICATION

C
\texttt{textport(left, right, bottom, top)}
\texttt{Screencoord left, right, bottom, top;}

FORTRAN
\texttt{subroutine textpo(left, right, bottom, top)}
\texttt{integer*4 left, right, bottom, top}

Pascal
\texttt{procedure textport(left, right, bottom, top: Screencoord);}

DESCRIPTION
textport allocates an area on the screen for the textport.

SEE ALSO
pagecolor, pagewritemask, textcolor, textwritemask, tpooff, tpon, viewport

NOTE
This command can be used only in immediate mode.
NAME
textwritemask - grants write permission for the textport

SPECIFICATION

C
    textwritemask(tmask)
    Colorindex tmask;

FORTRAN
    subroutine textwr(tmask)
    integer*4 tmask

Pascal
    procedure textwritemask(tmask: Colorindex);

DESCRIPTION
textwritemask grants write permission for text drawn in the textport. This
command does not affect text drawn with charstr.

SEE ALSO
pagecolor, pagewritemask, textcolor, textport, tpoft, tpon, writemask

NOTE
This command can be used only in immediate mode.
NAME

tie - ties two valuators to a button

SPECIFICATION

C

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

FORTRAN

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

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

DESCRIPTION

tie takes three arguments, a button b and two valuators v1 and v2. Whenever the button is queued, the valuators are also queued. If the button is queued, whenever the button changes state, three entries are made in the queue, recording the current state of the button and the current positions of each valuator. One valuator can be tied to a button by making v2 = 0. A button can be untied by making both v1 and v2 zero. v1 will appear before v2 in the event queue; both are preceded by b.

SEE ALSO

gobutton

NOTE

This command can be used only in immediate mode.
NAME
  tpoff - turns off the textport

SPECIFICATION
  C       tpoff()

  FORTRAN subroutine tpoff

  Pascal  procedure tpoff;

DESCRIPTION
  tpoff turns off the textport.

SEE ALSO
  pagecolor, pagewritemask, textcolor, textport, textwritemask, tpon

NOTE
  This command can be used only in immediate mode.
tpon

NAME
tpon - turns on textport

SPECIFICATION
C
tpon()

FORTRAN
subroutine tpon

Pascal
procedure tpon;

DESCRIPTION
tpon turns on the textport allocated by textport.

SEE ALSO
color, pagecolor, pagewritemask, textcolor, textport, textwritemask, tpooff

NOTE
This command can be used only in immediate mode.
NAME

translate - translates graphical primitives

SPECIFICATION

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

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

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

DESCRIPTION

translate places the object space origin at a given world coordinate point.
translate is a modeling command, and changes the current transformation
matrix. All objects drawn after translate is executed will be translated. Use
pushmatrix and popmatrix to limit the scope of the translation.

SEE ALSO

popmatrix, pushmatrix, rotate, scale
unqdevice

NAME

unqdevice - unqueues a device from the event queue

SPECIFICATION

C

unqdevice(v)
Device v;

FORTRAN

subroutine unqdev(v)
integer*4 v

Pascal

procedure unqdevice(v: Device);

DESCRIPTION

unqdevice removes the specified device from the list of those whose changes are automatically recorded in the event queue. If the device already has recorded events in the queue that have not been read, they remain there. qreset can be used to flush the event queue.

SEE ALSO

qdevice, qreset

NOTE

This command can be used only in immediate mode.
NAME
viewport - allocates an area of the screen for an image

SPECIFICATION
C
viewport(left, right, bottom, top)
Screencoord left, right, bottom, top;

FORTRAN
subroutine viewport(left, right, bottom, top)
integer*4 left, right, bottom, top

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

DESCRIPTION
The first step in defining the mapping from world coordinates to screen coordinates is to choose an area of the screen to display an image. This area is called the viewport and is specified in screen coordinates. The portion of world space described by one of the commands window, ortho, or perspective is mapped into the viewport. The arguments to the viewport command define a rectangular area on the screen by the left, right, bottom, and top coordinates. This command also loads the screenmask.

SEE ALSO
scrmask, getviewport, popviewport, pushviewport
NAME

window - defines a perspective projection transformation

SPECIFICATION

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

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

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

DESCRIPTION

window specifies the position and size of a rectangular viewing frustum in
terms of the distance to the surface closest to the eye (in the near clipping
plane), the boundaries of a rectangular region, and the distance to the far
clipping plane. The image will be projected with perspective onto the screen
area defined by viewport.

window loads a matrix onto the transformation stack, overwriting whatever
was there.

SEE ALSO

ortho, perspective, viewport

Version 2.1

IRIS User's Guide
NAME

writemask - grants write permission to available bitplanes

SPECIFICATION

C

writemask(wtm)
Colorindex wtm;

FORTRAN

subroutine writem(wtm)
integer*4 wtm

Pascal

procedure writemask(wtm: Colorindex);

DESCRIPTION

writemask shields selected bitplanes reserved for special uses from ordinary drawing commands. Its argument is a mask with one bit per available bitplane. Wherever there are ones in the writemask, the corresponding bits in the color will be written into the bitplanes. Zeros in the writemask mark bitplanes as read-only. These planes will not be changed, regardless of the bits in the color. RGBwritemask should be used in RGB mode.

SEE ALSO

RGBwritemask, color
writepixels

NAME

writepixels - paints a row of pixels on the screen

SPECIFICATION

C

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

FORTRAN

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

Pascal

procedure writepixels(n: Short; var colors: Colorarray);

DESCRIPTION

writepixels paints a row of pixels on the screen, specifying the number of pixels to paint and a color for each pixel. The current character position provides the starting location; it is updated to point to the pixel following the last one painted. The current character position becomes undefined if the updated pixel position is greater than XMAXSCREEN. Pixels are painted from left to right, and are clipped to the current screenmask. writepixels does not automatically wrap from one line to the next. It can be used in single and double buffer display modes. Use writeRGB in RGB mode.

SEE ALSO

scrmask, color, readpixels, writeRGB

NOTE

This command can be used only in immediate mode.
NAME

writeRGB - paints a row of pixels on the screen

SPECIFICATION

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

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

Pascal
procedure writeRGB(n: Short; var red, green, blue: RGBarray);

DESCRIPTION

writeRGB paints a row of pixels on the screen in RGB mode. It specifies the number of pixels to paint and a color for each pixel. The current character position provides the starting location; it is updated to point to the pixel following the last one painted. The current character position becomes undefined if the updated pixel position is greater than XMAXSCREEN. Pixels are painted from left to right, and are clipped to the current screenmask. writeRGB does not automatically wrap from one line to the next. It supplies a 24-bit RGB value (eight bits for each color) for each pixel. This value is written directly into the bitplanes.

SEE ALSO

scrmask, readRGB, RGBcolor, RGBwritemask, writepixels

NOTE

This command can be used only in immediate mode and RGB mode.
NAME

xfpt - transforms points

SPECIFICATION

C

xfpt(x, y, z)
Coord x, y, z;
xfpti(x, y, z)
Lcoord x, y, z;
xfpts(x, y, z)
Scoord x, y, z;
xfpt2(x, y)
Coord x, y;
xfpt2i(x, y)
Lcoord x, y;
xfpt2s(x, y)
Scoord x, y;
xfpt4(x, y, z, w)
Coord x, y, z, w;
xfpt4i(x, y, z, w)
Lcoord x, y, z, w;
xfpt4s(x, y, z, w)
Scoord x, y, z, w;

FORTRAN

subroutine xftp(x, y, z)
real x, y, z
subroutine xftp(x, y, z)
integer*4 x, y, z
subroutine xftsi(x, y, z)
integer*2 x, y, z
subroutine xfts(x, y, z)
subroutine xftp2(x, y)
real x, y
subroutine xftp2i(x, y)
integer*4 x, y
subroutine xftp2s(x, y)
integer*2 x, y
subroutine xfpt4(x, y, z, w)
real x, y, z, w
subroutine xfpt4i(x, y, z, w)
integer*4 x, y, z, w
subroutine xfpt4s(x, y, z, w)
integer*2 x, y, z, w

Pascal
procedure xfpt(x, y, z: Coord);
procedure xfpti(x, y, z: Icoord);
procedure xfpt5(x, y, z: Scoord);
procedure xfpt2(x, y: Coord);
procedure xfpt2i(x, y: Icoord);
procedure xfpt2s(x, y: Scoord);
procedure xfpt4(x, y, z, w: Coord);
procedure xfpt4i(x, y, z, w: Icoord);
procedure xfpt4s(x, y, z, w: Scoord);

DESCRIPTION

xfpt multiplies the specified point by the top matrix on the matrix stack and turns off the clippers and scalers in the Geometry Pipeline. In feedback mode, the 4-dimensional result of the multiplication is saved in the feedback buffer. In non-feedback mode, the command is ignored.

NOTE

This command can be used only in immediate mode.
NAME

zbuffer - starts or ends z-buffer operation

SPECIFICATION

C
zbuffer(bool)
Boolean bool;

FORTRAN
subroutine zbuffe(bool)
logical bool

Pascal
procedure zbuffer(bool: Boolean);

DESCRIPTION

zbuffer starts \((bool = TRUE)\) or ends \((bool = FALSE)\) z-buffer mode. In z-buffer mode, each pixel has an associated \(z\) value. When a pixel is to be drawn, a new \(z\) value is compared with the \(z\) value already associated with the pixel. If the new \(z\) value is less than or equal to the existing one (i.e., closer to the viewer), then a new color and \(z\) value for the pixel are stored in the bitplanes; otherwise the color and \(z\) value for the pixel are left unchanged. The result of drawing objects in this mode is an image in which all obscured surfaces are left undisplayed. \(z\) values range from 0x8000 to 0x7FFF on a 32-bitplane system, and 0x0 to 0x7FF on a 28-bitplane system. This range is set using the setdepth command. \(z\)-buffering is not effective on systems with less than 28 bitplanes. Because memory bandwidth is reduced on 60Hz monitors, performance is improved by blanking the screen during \(z\)-buffer drawing with the blankscreen command.

SEE ALSO

getzbuffer, zclear, setdepth, blankscreen

version 2.1 - 1 - iris user's guide
NAME
zclear - initializes the z-buffer to largest possible integer

SPECIFICATION
C
zclear()

FORTRAN
subroutine zclear

Pascal
procedure zclear;

DESCRIPTION
zclear initializes the z-buffer to the largest possible integer. This is most frequently done when z-buffer mode is entered to clear the buffer so that primitives drawn in z-buffer mode are only affected by each other.

SEE ALSO
zbuffer
Glossary

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 \( P_1, P_2, P_3, P_4 \) are 4D control points, then the spline determined by them is given by:

\[
C(t) = [1, t, t^2, t^3] \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}, \quad (0 \leq t \leq 1)
\]

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 \texttt{bbox} 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 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 callabj 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 `polarview` 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:

\[
A + (B - A) \cdot \left( \frac{c - a}{b - a} \right)
\]

**linestyle**: The 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:

\[ C(t) = (x(t), y(t), z(t)), \]

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:

\[ x = r \cos \Theta \]
\[ y = r \sin \Theta \]

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 \(-1 \leq x,y,z \leq 1\) to physical screen coordinates.

calendar 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.

calendar mask: 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 select 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 \( \mathbf{a} = (a_1, a_2, a_3) \) and \( \mathbf{b} = (b_1, b_2, b_3) \) are two 3D vectors, the vector product \( \mathbf{a} \times \mathbf{b} = (a_2 b_3 - a_3 b_2, a_3 b_1 - a_1 b_3, a_1 b_2 - b_1 a_2) \).

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.
Silicon Graphics, Inc.

Name: ____________________________
Title: ____________________________
Department: _______________________
Company: _________________________
Address: __________________________
Phone: ____________________________
Date: _____________________________

Reader's Comments
How could this manual be improved?

__________________________

Please list any errors, inaccuracies, or omissions you have found in this manual.

__________________________

Additional comments:

__________________________
BUSINESS REPLY CARD
PERMIT NO. 45 MOUNTAIN VIEW, CA

Silicon Graphics, Inc.
630 Clyde Court
Mountain View, California 94043

Attention: Technical Publications