

Graphics Library Programming Guide Volume II

Document Number 007-1702-020

Contributors

Written by Patricia McLendon

Illustrated by Dan Young, Howard Look, and Patricia McLendon

Edited by Steven W. Hiatt

Production by Derrald Vogt

Engineering contributions by John Airey, Kurt Akeley, Dan Baum, Rosemary Chang, Howard Cheng, Tom Davis, Bob Drebin, Ben Garlick, Mark Grossman, Michael Jones, Seth Katz, Phil Karlton, George Kong, Erik Lindholm, Howard Look, Rob Mace, Martin McDonald, Jackie Neider, Mark Segal, Dave Spalding, Gary Tarolli, Vince Uttley, and Rolf Van Widenfelt.

© Copyright 1992, Silicon Graphics, Inc.— All Rights Reserved

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

Restricted Rights Legend

Use, duplication, or disclosure of the technical data contained in this document by the Government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 52.227-7013 and/or in similar or successor clauses in the FAR, or in the DOD or NASA FAR Supplement. Unpublished rights reserved under the Copyright Laws of the United States. Contractor/manufacturer is Silicon Graphics, Inc., 2011 N. Shoreline Blvd., Mountain View, CA 94039-7311.

Graphics Library Programming Guide Document Number 007-1702-020

Silicon Graphics, Inc. Mountain View, California

Silicon Graphics, the Silicon Graphics logo, and IRIS are registered trademarks and IRIS Graphics Library, Geometry Engine, Geometry Pipeline, IRIX, 4D, MicroPixel, Personal IRIS, IRIS Indigo, XS, XS24, Elan, RealityEngine, and SkyWriter are trademarks of Silicon Graphics, Inc.

Ada is a registered trademark of the Ada Joint Program of the United States of America
Ethernet is a registered trademark of Xerox Corporation

OSF/Motif is a trademark of the Open Software Foundation

Spaceball is a trademark of Spatial Systems, Inc.

UNIX is a registered trademark of AT&T Bell Labs

X Window System and Athena Widgets are trademarks of the Massachusetts Institute of Technology.

Contents

Introduction	xxi
How to Use This Guide	xxii
How to Use the Sample Programs.....	xxii
Typographical Conventions	xxiii
What this Guide Contains.....	xxiii
How to Use the On-line Manual Pages.....	xxv
Using Man Pages from the Toolchest.....	xxv
Using Man Pages from an IRIX Shell	xxv
Suggestions for Further Reading	xxvi
1. Graphics Development Environment	1-1
1.1 Using the Graphics Library and System Libraries	1-1
1.1.1 System Libraries	1-2
1.1.2 Include Files	1-2
1.2 Using the X Window System.....	1-2
1.3 Programming in C.....	1-4
1.3.1 Using the ANSI C Standard.....	1-4
1.3.2 Compiling C Programs.....	1-4
1.4 GL Program Structure	1-5
1.4.1 Initializing the System	1-6
1.4.2 Getting Graphics Information	1-7
1.4.3 Global State Attributes	1-10
1.4.4 Exiting the Graphics Environment	1-10

2.	Drawing	2-1
2.1	Drawing with the GL.....	2-1
2.1.1	Vertex Subroutines.....	2-2
2.1.2	Points.....	2-7
2.1.3	Lines	2-8
2.1.4	Polygons	2-12
2.1.5	Point-Sampled Polygons	2-17
2.1.6	Meshes	2-21
2.1.7	Controlling Polygon Rendering	2-29
2.2	Old-Style Drawing	2-30
2.2.1	Basic Shapes	2-31
2.2.2	Nonrecommended Old-Style Subroutines	2-37
3.	Characters and Fonts	3-1
3.1	Drawing Characters	3-2
3.1.1	Determining the Current Character Position	3-2
3.1.2	Drawing Character Strings	3-3
3.1.3	Clipping Character Strings	3-3
3.1.4	Getting Character Information.....	3-4
3.2	Creating a Font	3-7
3.2.1	Defining the Font.....	3-9
3.2.2	Selecting the Font	3-12
3.2.3	Querying the System for Font Information	3-14
4.	Display and Color Modes	4-1
4.1	Color Display	4-2
4.1.1	Bitplanes	4-3
4.1.2	Dithering.....	4-4
4.2	RGB Mode	4-5
4.2.1	Setting and Getting the Current Color in RGB Mode.....	4-7
4.3	Gouraud Shading	4-8

4.4	Color Map Mode	4-15
4.4.1	Setting the Current Color in Color Map Mode.....	4-17
4.4.2	Getting Color Information in Color Map Mode.....	4-18
4.4.3	Gouraud Shading in Color Map Mode	4-19
4.4.4	Blinking.....	4-21
4.5	Onemap and Multimap Modes.....	4-22
4.6	Gamma Correction.....	4-23
5.	User Input	5-1
5.1	Event Handling	5-1
5.1.1	Queueing	5-3
5.1.2	Polling.....	5-7
5.2	Input Devices.....	5-10
5.2.1	Buttons	5-11
5.2.2	Valuators	5-12
5.2.3	Keyboard Devices	5-13
5.2.4	Window Manager Tokens.....	5-14
5.2.5	Spaceball™ Devices.....	5-15
5.2.6	Controlling Devices	5-16
5.3	Video Control.....	5-17
6.	Animation.....	6-1
6.1	Understanding How Animation Works	6-1
6.1.1	Single Buffer Mode	6-2
6.1.2	Double Buffer Mode	6-2
6.2	Creating an Animation.....	6-2
6.2.1	Setting Modes for Animation	6-3
6.2.2	Swapping Buffers	6-3
6.2.3	Swapping Multiple Buffers.....	6-3
6.2.4	Maximizing Animation Performance.....	6-6

7.	Coordinate Transformations	7-1
7.1	Coordinate Systems	7-2
7.2	Projection Transformations.....	7-4
7.2.1	Perspective Projection.....	7-4
7.2.2	Window Projection	7-9
7.2.3	Orthographic Projections	7-10
7.3	Viewing Transformations	7-12
7.3.1	Viewpoint in Polar Coordinates.....	7-13
7.3.2	Viewpoint along a Line of Sight.....	7-14
7.4	Modeling Transformations	7-16
7.4.1	Rotation.....	7-17
7.4.2	Translation.....	7-17
7.4.3	Scaling	7-18
7.5	Controlling the Order of Transformations	7-19
7.5.1	Current Matrix Mode (mmode)	7-19
7.6	Hierarchical Drawing with the Matrix Stack	7-20
7.7	Viewports, Screenmasks, and Scrboxes	7-26
7.8	User-Defined Transformations.....	7-29
7.9	Additional Clipping Planes	7-30
8.	Hidden-Surface Removal.....	8-1
8.1	z-buffering	8-2
8.1.1	Controlling z Values	8-7
8.1.2	Clearing the z-buffer and the Bitplanes Simultaneously	8-8
8.2	Using z-buffer Features for Special Applications.....	8-10
8.2.1	Drawing into the z-buffer.....	8-10
8.2.2	Alternative Comparisons and z-buffer Writemasks	8-14
8.3	Stenciling	8-16
8.4	Eliminating Backfacing Polygons	8-21
8.5	Alpha Comparison.....	8-22

9.	Lighting	9-1
9.1	Introduction to GL Lighting	9-2
	9.1.1 Color.....	9-2
	9.1.2 Reflectance.....	9-3
9.2	Setting Up GL Lighting	9-4
	9.2.1 Defining Surface Normals.....	9-5
	9.2.2 Defining Lighting Components	9-6
9.3	Binding Lighting Definitions.....	9-11
9.4	Changing Lighting Settings	9-12
9.5	Default Settings	9-13
9.6	Advanced Lighting Features	9-14
	9.6.1 Attenuation	9-14
	9.6.2 Spotlights.....	9-16
	9.6.3 Two-Sided Lighting	9-17
	9.6.4 Fast Updates to Material Properties	9-18
	9.6.5 Transparency	9-21
	9.6.6 Lighting Multiple GL Windows.....	9-21
9.7	Lighting Performance	9-22
	9.7.1 Restrictions on ModelView, Projection, and User-Defined Matrices.....	9-22
	9.7.2 Computational Considerations	9-23
9.8	Color Map Lighting	9-24
9.9	Sample Lighting Program.....	9-25
10.	Framebuffers and Drawing Modes	10-1
10.1	Framebuffers	10-1
	10.1.1 Normal Framebuffer	10-2
	10.1.2 Overlay Framebuffer	10-3
	10.1.3 Underlay Framebuffer.....	10-3
	10.1.4 Pop-up Framebuffer.....	10-3
	10.1.5 Left and Right Stereo Framebuffers.....	10-4
10.2	Drawing Modes.....	10-5

10.3	Writemasks.....	10-7
10.3.1	How Writemasks Work.....	10-8
10.3.2	Writemask Subroutines.....	10-9
10.3.3	Sample Writemask Programs.....	10-10
10.4	Configuring Overlay and Underlay Bitplanes.....	10-15
10.5	Cursor Techniques.....	10-17
10.5.1	Types of Cursors.....	10-17
10.5.2	Creating and Using Cursors.....	10-19
10.5.3	Cursor Subroutines.....	10-20
10.5.4	Sample Cursor Program.....	10-21
11.	Pixels.....	11-1
11.1	Pixel Formats.....	11-2
11.2	Reading and Writing Pixels Efficiently.....	11-3
11.2.1	Pixel Addressing.....	11-4
11.2.2	Reading Pixels.....	11-5
11.2.3	Writing Pixels.....	11-7
11.3	Using pixmode.....	11-10
11.3.1	Shifting Pixels.....	11-10
11.3.2	Expanding Pixels.....	11-11
11.3.3	Adding Pixels.....	11-11
11.3.4	Pixels Destined for the z-Buffer.....	11-12
11.3.5	Changing Pixel Fill Directions.....	11-12
11.4	Subimages within Images.....	11-13
11.5	Packing and Unpacking Pixel Data.....	11-14
11.6	Order of Pixel Operations.....	11-16
11.7	Old-Style Pixel Access.....	11-16
11.7.1	Reading Pixels.....	11-17
11.7.2	Writing Pixels.....	11-18

12.	Picking and Selecting	12-1
12.1	Picking	12-1
12.1.1	Defining the Picking Region.....	12-3
12.1.2	Using the Name Stack	12-4
12.2	Selecting.....	12-8
13.	Depth-Cueing and Atmospheric Effects	13-1
13.1	Depth-Cueing	13-1
13.1.1	Setting Up Depth-Cueing	13-2
13.1.2	Depth-Cueing in Colormap Mode.....	13-4
13.1.3	Depth-Cueing in RGBmode.....	13-6
13.1.4	Sample Depth-Cueing Program.....	13-7
13.2	Atmospheric Effects.....	13-10
13.2.1	Fog.....	13-11
13.2.2	Fog Characteristics	13-12
13.2.3	Fog Calculations	13-12
13.2.4	Fog Parameters	13-14
14.	Curves and Surfaces	14-1
14.1	Introduction to Curves	14-2
14.1.1	Parametric Curves.....	14-2
14.1.2	Polynomial Curves.....	14-5
14.1.3	Parametric Spline Curves.....	14-5
14.2	B-Splines	14-6
14.2.1	Control Points	14-6
14.2.2	Basis Functions	14-7
14.2.3	Knots	14-8
14.2.4	Weights	14-10
14.3	GL NURBS Curves.....	14-11
14.4	NURBS Surfaces	14-13
14.4.1	Geometric Surfaces	14-16
14.4.2	Color Surfaces	14-17
14.4.3	Texture Surfaces	14-17
14.5	Trimming NURBS Surfaces	14-18

14.6	NURBS Properties	14-21
14.7	Sample NURBS Program	14-22
14.8	Old-Style Curves and Surfaces.....	14-28
14.8.1	Old-Style Curves	14-28
14.8.2	Drawing Old-Style Curves	14-29
14.8.3	Drawing Old-Style Surfaces	14-33
15.	Antialiasing	15-1
15.1	Sampling Accuracy	15-1
15.1.1	Subpixel Positioning	15-4
15.1.2	How Subpixel Positioning Affects Lines	15-4
15.2	Blending.....	15-5
15.3	One-Pass Antialiasing—the Smooth Primitives	15-10
15.3.1	High-Performance Antialiased Points—pntsmooth	15-10
15.3.2	High-Performance Antialiased Lines—linesmooth	15-15
15.3.3	High-Performance Antialiased Polygons—polysmooth	15-22
15.4	Multipass Antialiasing with the Accumulation Buffer.....	15-30
15.4.1	Configuring the Accumulation Buffer	15-30
15.4.2	Using the Accumulation Buffer	15-31
15.5	Antialiasing on RealityEngine Systems	15-36
15.5.1	Multisample Antialiasing.....	15-37
15.5.2	Configuring Your System for Multisampling	15-37
15.5.3	How Multisampling Affects Rendering	15-42
15.5.4	Using Advanced Multisample Options	15-44

16.	Graphical Objects	16-1
16.1	Creating an Object.....	16-1
16.2	Working with Objects.....	16-4
16.2.1	Drawing an Object	16-4
16.2.2	Bounding Boxes.....	16-7
16.2.3	Editing Objects.....	16-8
16.2.4	Using Tags.....	16-9
16.2.5	Inserting, Deleting, and Replacing within Objects	16-10
16.2.6	Managing Object Memory	16-12
16.2.7	Mapping Screen Coordinates to World Coordinates	16-13
17.	Feedback	17-1
17.1	Feedback on IRIS-4D/GT/GTX Systems.....	17-4
17.2	Feedback on the Personal IRIS and IRIS Indigo	17-6
17.3	Feedback on IRIS-4D/VGX, SkyWriter, and RealityEngine Systems.....	17-7
17.4	Feedback Example	17-8
17.5	Additional Notes on Feedback.....	17-10
18.	Textures	18-1
18.1	Texture Basics	18-2
18.1.1	2-D Textures	18-3
18.1.2	3-D Textures.....	18-4
18.1.3	How to Set Up Texturing	18-6
18.2	Defining a Texture.....	18-6
18.2.1	Using Texture Components.....	18-7
18.2.2	Loading a Texture Array	18-9
18.2.3	Defining and Binding a Texture.....	18-11
18.2.4	Selecting the Texel SIze	18-13
18.3	Using Texture Filters	18-16
18.3.1	Minification Filters	18-17
18.3.2	Using Magnification Filters	18-21

18.4	Using the Sharpen and DetailTexture Features.....	18-23
18.4.1	Using the Sharpen Feature.....	18-23
18.4.2	Using DetailTextures	18-27
18.5	Texture Coordinates	18-32
18.5.1	Assigning Texture Coordinates Explicitly.....	18-32
18.5.2	Generating Texture Coordinates Automatically	18-33
18.5.3	Texture Lookup Tables.....	18-35
18.5.4	Improving Interpolation Results on VGX Systems	18-37
18.6	Texture Environments	18-39
18.7	Texture Programming Hints	18-42
18.8	Sample Texture Programs.....	18-45
19.	Using the GL in a Networked Environment.....	19-1
19.1	Introduction	19-1
19.1.1	Protocol.....	19-2
19.1.2	Writing GL Programs to Use Network Transparent Features	19-2
19.1.3	Establishing a Connection.....	19-4
19.1.4	Using rlogin.....	19-4
19.2	Limitations and Incompatibilities.....	19-5
19.2.1	The callfunc Routine	19-5
19.2.2	Pop-up Menu Functions.....	19-5
19.2.3	Interrupts and Jumps.....	19-5
19.3	Using Multiple Server Connections	19-6
19.3.1	Establishing and Closing Multiple Network Connections.....	19-6
19.3.2	Graphics Input.....	19-9
19.3.3	Local Graphics Data.....	19-9
19.3.4	Sample Program - Multiple Connections on a Local Host	19-9
19.4	Configuration of the Network Transparent Interface.....	19-12
19.4.1	inetd.....	19-12
19.4.2	dgld	19-13

19.5	Error Messages	19-13
19.5.1	Connection Errors	19-14
19.5.2	Client Errors	19-14
19.5.3	Server Errors	19-15
19.5.4	Exit Status	19-16
A.	Scope of GL Subroutines	A-1
B.	Global State Attributes	B-1
C.	Transformation Matrices	C-1
C.1	Translation.....	C-1
C.2	Scaling and Mirroring.....	C-1
C.3	Rotation.....	C-2
C.4	Viewing Transformations	C-3
C.5	Perspective Transformations	C-4
C.6	Orthographic Transformations	C-5
D.	GL Error Messages	D-1
E.	Using Graphics and Share Groups.....	E-1
	Index.....	Index-1

Figures

Figure 2-1	Simple Convex Polygon	2-12
Figure 2-2	Simple Concave Polygon	2-13
Figure 2-3	Another Simple Concave Polygon.....	2-13
Figure 2-4	Nonsimple Polygon	2-14
Figure 2-5	Another Nonsimple Polygon.....	2-14
Figure 2-6	Bowtie Polygon.....	2-14
Figure 2-7	Non-Point-Sampled Polygons	2-17
Figure 2-8	Point-Sampled Polygons	2-19
Figure 2-9	Point-Sampled Polygon with Edges Exactly on Pixel Centers	2-19
Figure 2-10	Point Sampling Anomaly	2-20
Figure 2-11	Simple Triangle Mesh.....	2-21
Figure 2-12	Example of <code>swaptmesh()</code> Construction.....	2-23
Figure 2-13	Another <code>swaptmesh()</code> Example.....	2-24
Figure 2-14	Mesh of Quadrilateral Strips	2-28
Figure 2-15	Equivalent T-mesh	2-28
Figure 2-16	Clipping Behavior of <code>PYM_LINE</code>	2-29
Figure 3-1	How Character Strings are Clipped	3-4
Figure 3-2	Bitmask for the Character A	3-7
Figure 3-3	Character Bitmap Created with <code>defrasterfont()</code>	3-8
Figure 4-1	Gouraud Shaded Triangle.....	4-10
Figure 7-1	Coordinate Systems	7-2
Figure 7-2	Frustum.....	7-6
Figure 7-3	Clipping Planes	7-8
Figure 7-4	The <code>window()</code> Projection Transformation	7-9

Figure 7-5	The <code>ortho()</code> Projection Transformation	7-10
Figure 7-6	The <code>polarview()</code> Viewing Transformation.....	7-13
Figure 7-7	The <code>lookat()</code> Viewing Transformation	7-14
Figure 7-8	Modeling Transformations	7-16
Figure 7-9	Effects of Sequence of Translations and Rotations.....	7-18
Figure 8-1	Overlapping Polygons.....	8-2
Figure 10-1	Writemask	10-8
Figure 10-2	Example Cursors	10-18
Figure 13-1	Depth-Cued Cube	13-1
Figure 13-2	Viewing Volume Showing Clipping Planes and a Depth-Cued Line	13-3
Figure 13-3	Color Ramp	13-4
Figure 14-1	Parametric Curve as a Mapping	14-3
Figure 14-2	Cross-plot	14-4
Figure 14-3	Spline.....	14-5
Figure 14-4	Influence of Control Points on a Curve	14-6
Figure 14-5	A B-Spline with Its Control Polygon.....	14-6
Figure 14-6	Moving a Control Point to Change the Shape of a B-Spline	14-7
Figure 14-7	Basis Functions for an Example Cubic B-Spline	14-8
Figure 14-8	Basis Functions for a Cubic B-Spline with a Multiple Interior Knot	14-8
Figure 14-9	B-Spline Relationships.....	14-9
Figure 14-10	Increasing the Weight of a Control Point Increases Its Influence.....	14-10
Figure 14-11	Representing a Circle with NURBS	14-12
Figure 14-12	Parametric NURBS Surface.....	14-13
Figure 14-13	Trimmed Surface	14-18
Figure 14-14	Trimming Curves	14-19
Figure 15-1	Antialiased Line.....	15-3
Figure 15-2	Example Multisample Patterns	15-37
Figure 15-3	Flexible Framebuffer Configuration.....	15-39
Figure 16-1	Sphere Defined as an Object	16-3
Figure 16-2	Drawing a Hierarchical Object.....	16-6
Figure 16-3	Bounding Boxes	16-7

Figure 17-1	Effects of Clipping on Feedback	17-3
Figure 18-1	Textures for Wood and Water	18-2
Figure 18-2	Fully Textured Scene	18-3
Figure 18-3	Mapping from 2-D Texture Space to 3-D Object Space	18-4
Figure 18-4	3-D Texture.....	18-4
Figure 18-5	Extracting an Arbitrary Planar Texture from a 3-D Texture Volume	18-5
Figure 18-6	Example of a Tree Created with a 2-component Texture....	18-8
Figure 18-7	Structure of a Texture Array.....	18-9
Figure 18-8	Texture Minification	18-16
Figure 18-9	Texture Magnification	18-16
Figure 18-10	Texture Minification	18-17
Figure 18-11	MIPmap	18-17
Figure 18-12	Texture Magnification	18-21
Figure 18-13	LOD Extrapolation Curves	18-25
Figure 18-14	Clamping the LOD Extrapolation.....	18-26
Figure 18-15	LOD Interpolation Curves	18-29

Tables

Table 1-1	System Types and Graphics Library Versions.....	1-8
Table 1-2	Tokens for Graphics Resource Inquiries.....	1-9
Table 2-1	Vertex Subroutines.....	2-3
Table 2-2	Sequence of Vertices in a Mesh	2-22
Table 2-3	Rectangle Subroutines	2-32
Table 2-4	Screen Box Subroutines	2-34
Table 2-5	Circle Subroutines	2-34
Table 2-6	Arc Subroutines.....	2-36
Table 2-7	Old-Style Point Subroutines	2-38
Table 2-8	Old-Style Move and Draw Subroutines.....	2-39
Table 2-9	Old-Style Filled Polygon Move and Draw Subroutines.....	2-40
Table 2-10	Old-Style Polygon and Filled Polygon Subroutines	2-41
Table 3-1	cmov () Subroutines	3-3
Table 4-1	Bitplane Configurations of Silicon Graphics Workstations.....	4-3
Table 4-2	The Color (c) Family of Subroutines	4-7
Table 4-3	Default Color Map	4-16
Table 5-1	Class Ranges in the Device Domain	5-10
Table 5-2	Input Buttons	5-11
Table 5-3	Input Valuators.....	5-12
Table 5-4	Window Manager Event Tokens	5-14
Table 5-5	Spaceball Input Buttons	5-15
Table 5-6	Monitor Types	5-19
Table 5-7	Video Register Values.....	5-20
Table 5-8	Live Video Digitizer Commands	5-21

Table 8-1	Maximum and Minimum z-buffer Values.....	8-8
Table 8-2	Values of <code>zfunction()</code> for Personal IRIS <code>czclear()</code>	8-9
Table 10-1	Overlay and Underlay Bitplane Configurations	10-16
Table 11-1	Hints for <code>readdisplay()</code>	11-5
Table 15-1	Blending Factors.....	15-6
Table 15-2	Tokens for Selecting Accumulation Multisample Patterns.....	15-45
Table 17-1	IRIS-4D/G/GT/GTX Feedback Data.....	17-4
Table 18-1	Texture Components	18-7
Table 18-2	Texture Image Array Format.....	18-10
Table 18-3	Texture Component Configuration for Different Texel Sizes	18-14
Table 18-4	Formulas for Computing DetailTexture Filters	18-30
Table 18-5	The <code>t()</code> Subroutine	18-33
Table 18-6	Texture Look-up Table Actions.....	18-36
Table 18-7	<code>TV_MODULATE</code> Equations	18-40
Table 18-8	<code>TV_BLEND</code> Equations.....	18-40
Table 18-9	<code>TV_DECAL</code> Equations.....	18-41
Table 19-1	Error Values	19-14
Table 19-2	DGL Client Exit Values	19-15
Table 19-3	DGL Server Exit Value	19-16
Table A-1	GL State Types	A-1
Table A-2	Scope of GL Subroutines	A-2
Table B-1	Default Color Map Values	B-1
Table B-2	Keys to Information in Table B-3	B-2
Table B-3	Global State Attribute Defaults	B-2
Table D-1	GL Error Messages and Probable Causes.....	D-1

Depth-Cueing and Atmospheric Effects

This chapter introduces you to two techniques that contribute to the perception of 3-D in your scenes: depth-cueing and atmospheric effects.

- Section 13.1, “Depth-Cueing,” describes how to use depth-cueing in RGB mode and in colormap mode.
- Section 13.2, “Atmospheric Effects,” shows you how to create atmospheric effects such as fog and haze.

13.1 Depth-Cueing

When you look at objects in the real world, it is your eye’s ability to perceive depth—*depth perception*, that makes you aware of the 3-D nature of objects and lets you judge the relative distance of objects. The illusion of depth perception can be created on the 2-D screen by *depth-cueing* images. Depth-cueing modifies an object’s color based on its distance from the viewer. Figure 13-1 shows how a cube looks when it is depth-cued.

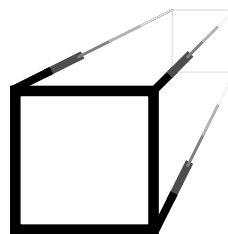


Figure 13-1 Depth-Cued Cube

Two methods of depth-cueing are presented in this chapter: color replacement and color blending.

Color replacement makes objects closer to the viewer brighter than those far away from the viewer. Depth-cueing makes an image appear 3D by replacing the color of all points, lines, and polygons with colors determined by their z values.

Color blending blends true object color with another color, where the blend ratio is determined by the depth of the object. You can use color blending for depth-cueing, but color blending is better known as a technique for simulating atmospheric phenomena such as fog and smoke. (See Section 13.2)

Note: Depth-cueing and lighting cannot be used simultaneously.

13.1.1 Setting Up Depth-Cueing

To set up depth-cueing in your GL application:

1. Set the proper modes:
`shademodel(GOURAUD);` – this is the default
2. Define a range of z values that describes the viewing volume that is subject to depth-cueing.
3. Clear the z-buffer to the maximum z value and clear the screen color to the background color.
4. Turn depth-cueing on.
5. Specify a mapping of z values to color.
6. Draw objects that are to be depth-cued.

Defining the Boundaries for Depth-Cueing

You need to tell the GL where in your viewing volume you want objects to be depth-cued. Doing so establishes a reference for mapping the maximum and minimum color intensities. The near and far clipping planes establish the reference for the color mapping.

Figure 13-2 shows how the minimum and maximum z values are mapped to the brightest and dimmest colors within a viewing volume.

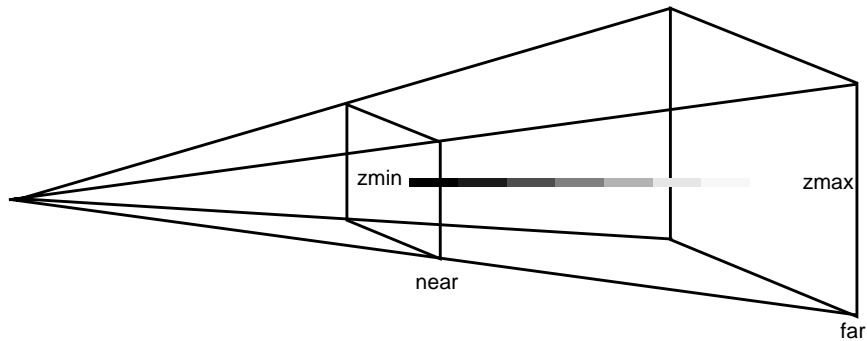


Figure 13-2 Viewing Volume Showing Clipping Planes and a Depth-Cued Line

Use the `lsetdepth()` subroutine to define the *near* and *far* z values that form the boundary z values used for depth-cueing within a viewing volume. The C specification for `lsetdepth()` is:

```
void lsetdepth(long near, long far)
```

The valid range of *near* and *far* depends on the state of the `GLC_ZRANGEMAP` compatibility mode you set. You use the `glcompat()` subroutine with the argument `GLC_ZRANGEMAP` to set the compatibility:

```
glcompat(GLC_ZRANGEMAP)
```

If `glcompat(GLC_ZRANGEMAP)` is set to 0, the valid range for *near* and *far* depends on the graphics hardware: the z minimum is the value returned by `getgdesc(GD_ZMIN)` and the z maximum is the value returned by `getgdesc(GD_ZMAX)`. If `GLC_ZRANGEMAP` is set to 1, the minimum is 0x0 and the maximum is 0x7FFFFFFF, and this range is mapped to whatever range the hardware supports. You should always explicitly set `glcompat(GLC_ZRANGEMAP)` because its default state depends on the machine type.

Turning Depth-Cueing On/Off

Use the `depthcue()` subroutine to turn depth-cue mode on and off. The ANSI C specification for `depthcue` is:

```
void depthcue(Boolean mode)
```

When you specify `TRUE`, all lines, points, characters, and polygons that the system draws are depth-cued. When you specify `FALSE`, depth-cue mode is off. Rendering in depth-cue mode may be somewhat slower, so turn off depth-cueing when you don't need it.

Querying the System for Depth-Cueing Mode

Use the subroutine `getdcm()` to query the system about whether depth-cueing is on or off. The ANSI C specification for `getdcm()` is:

```
boolean getdcm(void)
```

`TRUE` means depth-cue mode is on; `FALSE` means depth-cue mode is off.

Specify a mapping of `z` values to color by using either the `lshaderange()` or `lRGBrange()` command. In colormap mode, use `lshaderange()` to describe a mapping from `z` values to color index values. In RGB mode, use `lRGBrange()` to describe a mapping from `z` values to RGB values.

13.1.2 Depth-Cueing in Colormap Mode

When you use depth-cueing in color map mode, the GL uses the colors that you define in the `z` value mapping when it draws the geometry. You need to create a color ramp for depth-cueing in color map mode. Figure 13-3 shows a typical color ramp.

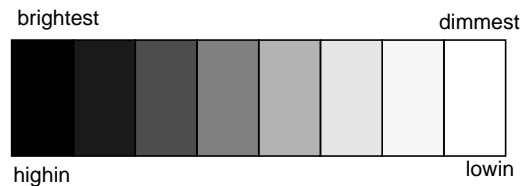


Figure 13-3 Color Ramp

Create a color ramp by specifying the color values at each end of the ramp, and how the colors are incremented.

Use the `mapcolor()` subroutine to load your color ramp into the color map. (See Chapter 4 for information on using `mapcolor()`.)

Use the `lshaderange()` subroutine to define the mapping from z value to color.

The ANSI C specification for `lshaderange()` is:

```
void lshaderange(Colorindex lowin, Colorindex highin,  
                long znear, long zfar)
```

Specify the low-intensity color map index (*lowin*) and the high-intensity color map index (*highin*) in the `lshaderange()` subroutine. These values are mapped to the near and far z values that you specify for *znear* and *zfar*.

`lshaderange()` defines the entire transformation range. The brightest color is mapped to *znear* and the dimmest color is mapped to *zfar*. The color of lines or points extending beyond *znear* and *zfar* are clamped to the brightest and dimmest values respectively. Screen z values nearer than *znear* map to *highin* and screen z values farther than *zfar* map to *lowin*.

The values of *znear* and *zfar* should correspond to or lie within the range of z values specified by `lsetdepth()`. If *near* < *far*, then *znear* should be less than *zfar*. If *near* > *far*, then *znear* should be greater than *zfar*. In other words, the range [*near*, *far*] that you define in `lsetdepth()` should bound the range [*znear*, *zfar*] that you define in `lshaderange()`.

The entries for the color map between *lowin* and the *highin* 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 color intensity along the line is linearly interpolated from the intensities of its endpoints, which are determined from their z values.

You can achieve higher resolution if the near and far clipping planes bound the object as closely as possible.

The following equation yields the color map index for a point with a z coordinate of z . Note that this equation yields a nonlinear mapping when z is outside the range of $[z_{near}, z_{far}]$. Because depth-cued lines are linearly interpolated between endpoints, an endpoint outside the range of $[z_{near}, z_{far}]$ can result in an undesirable image.

$$color_z = \begin{cases} highin, & \text{if } (z \leq z_{near}) \\ \left(\frac{lowin - highin}{z_{far} - z_{near}} \right) (z - z_{near}) + highin, & \text{if } (z_{near} \leq z \leq z_{far}) \\ lowin, & \text{if } (z_{far} \leq z) \end{cases} \quad \text{(EQ 13-1)}$$

13.1.3 Depth-Cueing in RGBmode

When you use depth-cueing in RGB mode, the GL uses the colors that you define in the z value mapping when draws the geometry.

You use the `lRGBrange()` subroutine to set the range of colors to use for depth-cueing in RGB mode. The C specification for `lRGBrange` is:

```
void lRGBrange (short rmin, short gmin, short bmin, short
rmax, short gmax, short bmax, long zmin, long zmax)
```

Specify the minimum and maximum values to be stored in the color bitplanes, and the near and far z values to which the colors are mapped. $rmin$ and $rmax$ are the minimum and maximum values stored in the red bitplanes. Likewise, $gmin$, $gmax$, $bmin$, and $bmax$ define the minimum and maximum values stored in the green and blue bitplanes, respectively. z_{near} and z_{far} define the z values that are mapped linearly into the RGB range. z values nearer than z_{near} are mapped to $rmax$, $gmax$, and $bmax$; z values farther than z_{far} are mapped to $rmin$, $gmin$, and $bmin$.

13.1.4 Sample Depth-Cueing Program

This sample program, *depthcue.c*, draws a cube filled with points that rotates as you move the mouse. Because 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.

```
#include <stdio.h>
#include <math.h>
#include <gl/gl.h>
#include <gl/device.h>

#define RGB_BLACK 0x000000

#define X      0
#define Y      1
#define Z      2
#define XY     2
#define XYZ    3

#define CORNERDIST 1.8          /* center to furthest corner of cube */
#define EYEDIST 3*CORNERDIST   /* center to eye */

#define NUMPOINTS 100

float points[NUMPOINTS][XYZ];
long corner[8][XYZ] = {
    {-1, -1, -1},
    {-1, 1, -1},
    { 1, -1, -1},
    { 1, 1, -1},
    {-1, -1, 1},
    {-1, 1, 1},
    { 1, -1, 1},
    { 1, 1, 1}
};

int edge[12][2] = {
    {0, 1}, {1, 3}, {3, 2}, {2, 0},
    {4, 5}, {5, 7}, {7, 6}, {6, 4},
    {0, 4}, {1, 5}, {2, 6}, {3, 7},
};

void drawcube()
{
    int i;
```

```

        for (i = 0; i < 12; i++) {
            bgnline();
                v3i(corner[edge[i][0]]);
                v3i(corner[edge[i][1]]);
            endline();
        }
    }

void drawpoints()
{
    int i;

    bgnpoint();
        for (i = 0; i < NUMPOINTS; i++)
            v3f(points[i]);
    endpoint();
    drawcube();
}

main()
{
    long maxscreen[XY];
    Device mdev[XY];
    short mval[XY];
    float rotang[XY];
    short val;
    int i;
    if (getgdesc(GD_BITS_NORM_DBL_RED) == 0) {
        fprintf(stderr, "Double buffered RGB not available on this machine \n");
        return 1;
    }
    prefsiz(400, 400);
    winopen("depthcue");
    doublebuffer();
    RGBmode();
    gconfig();
    cpack(RGB_BLACK);
    clear();
    swapbuffers();
    qdevice(ESCKEY);
    maxscreen[X] = getgdesc(GD_XPMAX) - 1;
    maxscreen[Y] = getgdesc(GD_YPMAX) - 1;
    mdev[X] = MOUSEX;
    mdev[Y] = MOUSEY;
    mmode(MVIEWING);
    window(-CORNERDIST, CORNERDIST, -CORNERDIST, CORNERDIST,
           EYEDIST, EYEDIST + 2*CORNERDIST);
}

```

```

lookat(0.0, 0.0, EYEDIST + CORNERDIST, 0.0, 0.0, 0.0, 0);
/* map the current machine's z range to 0x0 -> 0x7fffff */
glcompat(GLC_ZRANGEMAP, 1);
/* set up the mapping of screen z values to cyan intensity */
lRGBrange(0, 15, 15, 0, 255, 255, 0x0, 0x7fffff);
/* have screen z values control the color */
depthcue(TRUE);
/* generate random points */
for (i = 0; i < NUMPOINTS; i++) {
    points[i][X] = 2.0 * (drand48() - 0.5);
    points[i][Y] = 2.0 * (drand48() - 0.5);
    points[i][Z] = 2.0 * (drand48() - 0.5);
}
while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
    cpack(RGB_BLACK);
    clear();
    getdev(XY, mdev, mval);
    rotang[X] = 4.0 * (mval[Y] - 0.5 * maxscreen[Y]) / maxscreen[Y];
    rotang[Y] = 4.0 * (mval[X] - 0.5 * maxscreen[X]) / maxscreen[X];
    rot(rotang[X], 'x');
    rot(rotang[Y], 'y');
    drawpoints();
    swapbuffers();
}
gexit();
return 0;
}

```

13.2 Atmospheric Effects

You can create a variety of atmospheric effects on IRIS Indigo, IRIS-4D/VGX, SkyWriter, and RealityEngine systems. Atmospheric detail adds realism to visual simulations and provides interesting depth perception effects. You can simulate fog, smoke, haze and air pollution by varying the color and density of the atmosphere in your scene. In this section, the term “fog” is used to mean any of these atmospheric conditions.

The GL creates atmospheric effects by modifying the color of the objects in your scene based on their distance from the viewer. Object color is blended with fog color to determine the apparent color perceived by the viewer.

When fog is present, objects at close range to the viewer appear in true color. True color is the color an object would be in the absence of fog—that is, the color of the object computed after lighting, shading, and texture mapping (if any). Distant objects look “washed out” as the object color is blended with the fog color, which gives the appearance of seeing the object “through” the fog. There is a certain point in the distance where the fog completely obscures the object.

As an example, consider looking down the runway at an approaching airplane. On a clear sunny day, the airplane is seen in full detail, limited only by your visual acuity (the resolution of your eye). On a foggy day, your view of the airplane is impaired and its apparent color is a combination of its true color and the fog color. As the airplane approaches, your eye begins to detect its true color.

Depth-cueing is another effect possible with fog. When you blend an object’s true color with the scene background color, an object moving away from the viewer appears to fade into the background. An advantage of this method over using `depthcue()`/`lshaderange()` is that it is independent of the color of the viewed objects—you have to call `lshaderange()` each time the object color changes, whereas `fogvertex()` need only be called when the background color changes.

13.2.1 Fog

You define and enable fog with the `fogvertex()` subroutine. You call `fogvertex()` once to set up the fog parameters, then you turn the fog on with another call to `fogvertex()`. Fog is currently available only on Indigo, VGX, SkyWriter, and RealityEngine systems, so you should call `getgdesc(GD_FOGVERTEX)` to determine the fog capability of the machine when writing fog applications.

To use fog in your application:

1. Use `getgdesc()` to determine the fog capabilities of your machine.
2. Set up the proper modes for fog:

```
RGBmode()  
mmode(MVIEWING)
```

Note: Remember to call `gconfig()` if you change to RGB mode from colormap mode.

3. Define the fog characteristics.

```
fogvertex(mode, params)
```

Note: Defining a fog effect does not enable it. This must be done with a separate call.

4. Turn fog on.

```
fogvertex(FG_ON, dummy)
```

The ANSI C specification for `fogvertex()` is:

```
void fogvertex(long mode, float *params);
```

You use the mode argument to indicate whether you are defining, enabling, or disabling fog effects. You specify the fog characteristics in the *params* array. *params* is an array of floating point values.

13.2.2 Fog Characteristics

Fog characteristics define the color and density of the fog. You can specify *uniformly distributed fog*, where the fog has a uniform density throughout, or you can specify *linearly blended (interpolated) fog*, where the fog begins at a certain point and becomes so dense that it is opaque in the distance.

For uniformly distributed fog, you specify a fog density between 0.0 and 1.0 and the fog color. A fog density of 0.0 represents no fog at all—the object's apparent color is the same as its true color. Increasing positive values increase the fog density. The maximum fog density is 1.0. For a fog density of 1.0, fog totally obscures the true color of the viewed object at a distance of one unit in eye coordinates. This is the reference to which fog density values are normalized.

The proportion of the object's true color that contributes to the apparent color is called the *blend factor*. When you enable fog effects, the blend factor is computed at the vertices of each graphics primitive. The vertex blend factors are then interpolated to determine the blend factor at the interior pixels of the graphics primitive.

SkyWriter and RealityEngine systems allow you to specify per-pixel fog calculations, which is more accurate than interpolation. To maximize the accuracy of the fog, minimize the ratio of the distance to the far clipping plane to the distance to the near clipping plane. In other words, you specify near and far such that the near and far clipping planes are as close together as possible. In addition, you need to specify the maximum `lsetdepth()` range for your machine by calling `lsetdepth(getgdesc(GD_ZMIN), getgdesc(GD_ZMAX))`.

13.2.3 Fog Calculations

The blend factor (*fog*) is calculated according to one of the three equations that follow. The first two equations calculate *fog* in eye coordinates for uniformly distributed fog. The third equation calculates fog varying linearly with distance, given the distance at which the fog begins and the distance at which the fog becomes opaque. Each of these methods allow you to indicate per-vertex or per-pixel fog calculations.

Exponential fog: (EQ 13-2)

$$fog = (1 - e)^{(5.5 \cdot density \cdot Z_{eye})}$$

Exponential-squared fog: (EQ 13-3)

$$fog = (1 - e)^{(-5.5 \cdot (density \cdot Z_{eye})^2)}$$

Linear fog: (EQ 13-4)

$$fog = 1 - \frac{(end_fog + Z_{eye})}{(end_fog - start_fog)}$$

where:

fog is the computed fog blending factor ($0 \leq fog \leq 1$).

density is the fog density.

Z_{eye} is the eye space Z coordinate (always negative) of the pixel or vertex being fogged.

start_fog is the distance from the viewer where the fog begins to appear.

end_fog is the distance from the viewer where the fog becomes opaque.

The pixel color, *C_p*, is combined with fog color, *C_f*, to give apparent color, *C*:

$$C = C_p \cdot (1 - fog) + C_f \cdot fog \quad \text{(EQ 13-5)}$$

where:

fog is the computed fog blending factor, ($0 \leq fog \leq 1$).

C is the resultant color.

C_p is the incoming pixel color, which may be Gouraud or flat shaded and possibly textured.

C_f is the fog color.

13.2.4 Fog Parameters

Based on the effect you want to achieve, you enter one of the following symbolic constants for *mode* in `fogvertex()`:

<code>FG_VTX_EXP</code>	Fog is computed at each vertex of the primitive (EQ 13-2).
<code>FG_PIX_EXP</code>	Fog is computed at each pixel of the primitive (EQ 13-2).
<code>FG_VTX_EXP2</code>	Fog is computed at each vertex of the primitive (EQ 13-3).
<code>FG_PIX_EXP2</code>	Fog is computed at each pixel of the primitive, (EQ 13-3).
<code>FG_VTX_LIN</code>	Fog is computed at each vertex of the primitive (EQ 13-4).
<code>FG_PIX_LIN</code>	Fog is computed at each pixel of the primitive (EQ 13-4).

To enable or disable fog effects, use the following:

<code>FG_ON</code>	Enable the previously defined fog effect.
<code>FG_OFF</code>	Disable fog effects. This is the default.

You specify four floating point values in the *params* array for uniformly distributed fog (`FG_VTX_EXP`, `FG_PIX_EXP`, `FG_VTX_EXP2`, or `FG_PIX_EXP2`):

<i>density</i>	Density(thickness) of fog ($0.0 \leq \textit{density} \leq 1.0$).
<i>r</i>	Red component of fog ($0.0 \leq r \leq 1.0$).
<i>g</i>	Green component of fog ($0.0 \leq g \leq 1.0$).
<i>b</i>	Blue component of fog ($0.0 \leq b \leq 1.0$).

You specify five values in the *params* array for linearly blended fog (`FG_VTX_LIN`, or `FG_PIX_LIN`):

<i>start_fog</i>	Distance in eye coordinates to start of fog.
<i>end_fog</i>	Distance in eye coordinates where fog becomes completely opaque.
<i>r</i>	Red component of fog ($0.0 \leq r \leq 1.0$).
<i>g</i>	Green component of fog ($0.0 \leq g \leq 1.0$).
<i>b</i>	Blue component of fog ($0.0 \leq b \leq 1.0$).

Chapter 14

Curves and Surfaces

This chapter describes how to draw curves and surfaces using *Non-uniform Rational B-splines (NURBS)*. NURBS are useful for creating the types of smooth curves and surfaces that you see on airplane wings, cars, and machinery. You can render NURBS with color, lighting, and texture. You can also cut holes in the interior of a NURBS surface to create more complex surfaces.

- Section 14.1, “Introduction to Curves,” develops the background and terminology for curves and surfaces.
- Section 14.2, “B-Splines,” discusses characteristics of B-Splines.
- Section 14.3, “GL NURBS Curves,” tells you how to draw NURBS curves.
- Section 14.4, “NURBS Surfaces,” tells you how to draw NURBS surfaces.
- Section 14.5, “Trimming NURBS Surfaces,” tells you how to cut holes in NURBS surfaces.
- Section 14.6, “NURBS Properties,” describes how NURBS are rendered.
- Section 14.8, “Old-Style Curves and Surfaces,” describes the GL method of defining curves and surfaces that was used in previous releases of the software. This section is included for compatibility only—all new development should use the GL NURBS method.

The GL draws NURBS curves by efficiently approximating them with line segments, and it draws NURBS surfaces by efficiently approximating them with polygon meshes. This means you can use the GL commands for transformations, lighting, and hidden-surface removal to control how NURBS curves and surfaces are drawn, just as you do with other GL primitives.

To see an interactive NURBS curve demonstration, use the Demos Toolchest on your workstation to look at *Curves* and *NURBS* under *GL Demos*.

You can develop an intuitive understanding about NURBS by studying the text and accompanying illustrations presented in this chapter. For a more rigorous mathematical treatment, consult the “Suggestions for Further Reading” at the end of this chapter.

14.1 Introduction to Curves

This section discusses curves. First, basic curve information is presented, then B-Splines are discussed, and finally, NURBS are introduced. Once understood, principles learned about curves are easily extended to surfaces.

14.1.1 Parametric Curves

The most commonly used representation for curves (and for surfaces) is the *parametric* form, which is the one used in the GL; it is discussed in detail in the next section. You may be aware of other forms such as *implicit* or *algebraic*. An implicit or algebraic representation is of the form:

$$f(x, y) = 0 \quad (\text{EQ 14-6})$$

The following *explicit* representation is an alternative to the implicit form:

$$y = f(x) \quad (\text{EQ 14-7})$$

In this case, x is the independent variable and y is the dependent variable.

In the parametric representation, the coordinate functions x, y , (and z) are all explicit functions of a *parameter*. In this chapter, this parameter is called u .

In Figure 14-1, the function $\vec{F}(u)$ maps points from a subset of the real line \mathfrak{R} , to model coordinates (which can be 2-D, 3-D, or 4-D).

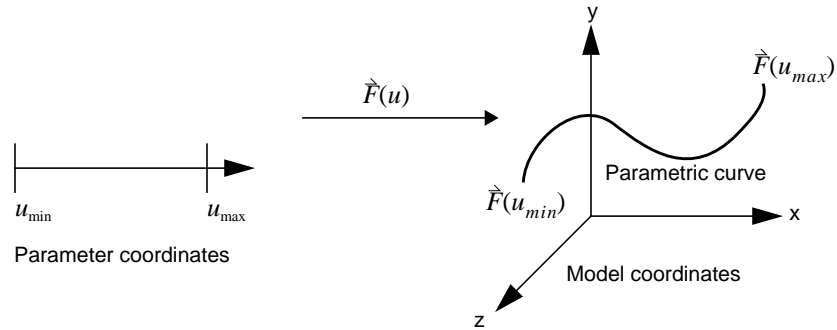


Figure 14-1 Parametric Curve as a Mapping

Call this vector-valued function the curve \vec{C} , which is defined for values of u in a given subset of the real line:

$$\vec{C}(u) = (x(u), y(u), z(u)), \quad \text{where } u \in [u_{\min}, u_{\max}] \in \mathfrak{R} \quad (\text{EQ 14-8})$$

As u takes on values from u_{\min} to u_{\max} , a *parametric curve* is traced out.

Each coordinate on the curve is a function of the parameter u .

Let u^* represent a specific value of u . The coordinates of the curve at that value of u are obtained by evaluating the functions x , y , and z at u^* :

$$\vec{C}(u^*) = (x^*, y^*, z^*), \quad \text{where } x^* = x(u^*), y^* = y(u^*), z^* = z(u^*) \quad 4-9$$

Figure 14-2 is called a *cross-plot*. A cross-plot helps you visualize how values from the real line are mapped to a parametric curve.

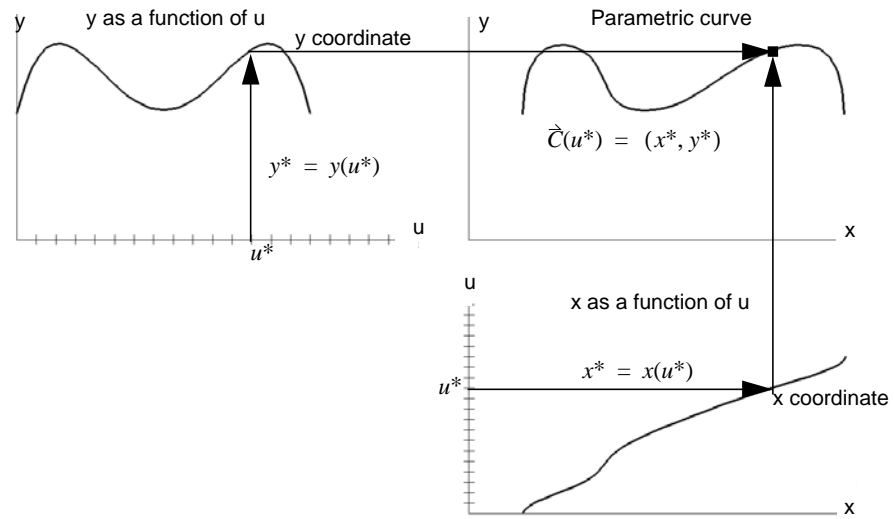


Figure 14-2 Cross-plot

The upper right graph in Figure 14-2 is a plot of a curve in 2-D model coordinates (x,y) . The upper left graph is a plot of the y coordinate as a function of the parameter u : $(u, y(u))$. Likewise, the lower right graph is a plot of x as a function of u that is drawn with u on the vertical axis $(x(u), u)$.

For a particular value of u , u^* , the corresponding $y(u^*)$ and $x(u^*)$ are found, as shown by the arrows from the u axis to the functions. By drawing horizontal and vertical lines as shown, the point (x^*,y^*) is located on the parametric curve.

14.1.2 Polynomial Curves

Curves can be represented mathematically as *polynomials*. The equations below, showing x and y as functions of u are examples of *cubic polynomials*. The *degree* of the polynomial is 3, because the highest exponent is 3.

(EQ 14-10)

$$x(u) = a_0 + a_1u + a_2u^2 + a_3u^3$$

$$y(u) = b_0 + b_1u + b_2u^2 + b_3u^3$$

The a_i 's and b_i 's in the polynomials above are called *coefficients*, also known as *control points*. Notice that the total number of coefficients in each polynomial is one greater than the degree. This number is also called the *order* of the polynomial. Order is equal to the degree plus one, so these are polynomials of order 4.

Putting the two polynomials into the parametric representation of the curve \vec{C} , gives you a *parametric cubic polynomial* curve:

$$\vec{C}(u) = (x(u), y(u)) \quad (\text{EQ 14-11})$$

14.1.3 Parametric Spline Curves

You can join polynomials together to make a *piecewise polynomial*, which gives you more flexibility in shaping the curve. This piecewise polynomial is called a *spline*. The order of a spline is defined as the maximum order of the component polynomial pieces. Generally, each polynomial has the same order. If necessary, you can elevate the order of the lower degree polynomials to the order of the highest order piece, using a process called *degree elevation* (See "B-Spline Basics" in *Curves and Surfaces for Computer Aided Geometric Design*). The breakpoints where the polynomial pieces are joined are called *knots*.

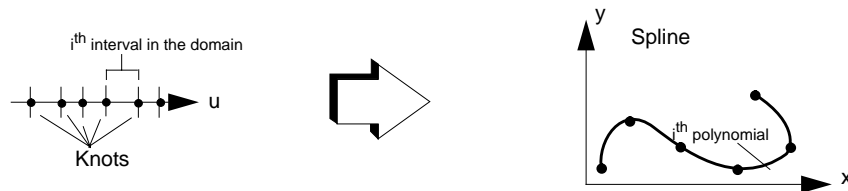


Figure 14-3 Spline

14.2 B-Splines

A *B-Spline* is a compact representation for piecewise polynomials. In the previous section, you learned that the knots of a spline tell you where one polynomial segment ends and a new one begins. This section describes how to shape the curve segments.

14.2.1 Control Points

Associated with a B-Spline is a set of points called *control points*. You use control points to shape the curve. Control points are like magnets—they pull the curve into a certain shape (see Figure 14-4).

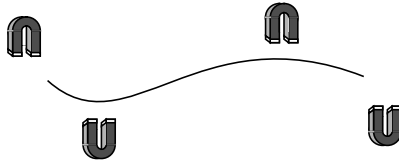


Figure 14-4 Influence of Control Points on a Curve

Connecting the control points in order forms a *control polygon*. The B-Spline lies close to its control polygon, as shown in Figure 14-5.

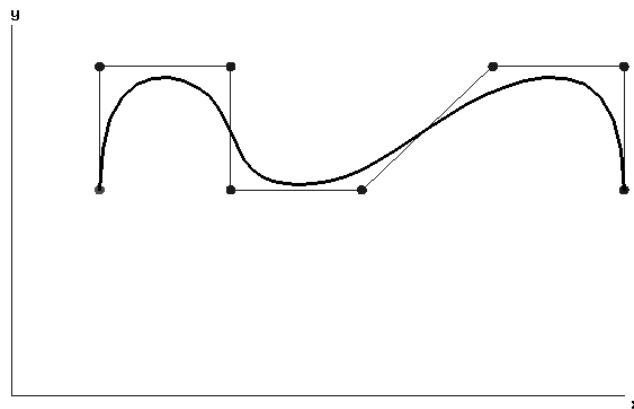


Figure 14-5 A B-Spline with Its Control Polygon

Moving a control point influences the shape of the curve segment near it. Figure 14-6 shows how the shape of the B-Spline changes when you move one of its control points.

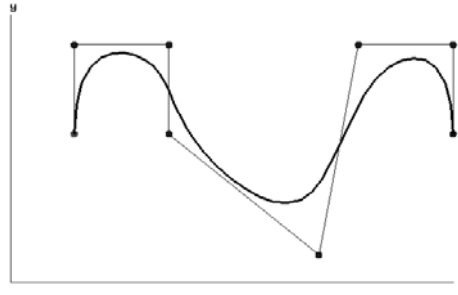


Figure 14-6 Moving a Control Point to Change the Shape of a B-Spline

The curve is really a weighted average of the influence of the control points. The notation below represents this concept mathematically:

$$\vec{C}(u) = \sum \vec{d}_i B_i(u) \quad (\text{EQ 14-12})$$

In this representation, the d_i 's are the control points and the $B_i(u)$'s are known as *basis functions*.

14.2.2 Basis Functions

Basis functions determine *how much* the control points influence the curve, *which* control points influence the curve and *where* the curve is influenced by each control point. These rules govern basis functions used in the GL:

1. There is a basis function corresponding to each control point.
2. Basis functions are positive (they attract rather than repel the curve).
3. The curve is defined only where *order* number of basis functions are defined.
4. At most, only *order* number of basis functions are non-zero at a time.
5. At any u in the domain, the basis functions add up to exactly 1.
6. The number of knots equals the number of control points plus the order.
7. The knots are given in a non-decreasing sequence.

Figure 14-7 shows an example of the basis functions for a cubic B-Spline. The basis functions in this example are uniformly spaced and are identical, except for a translation. Since the order is 4, the curve is defined only for values of u where all four basis functions are defined (nonzero), within the shaded area.

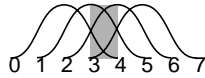


Figure 14-7 Basis Functions for an Example Cubic B-Spline

14.2.3 Knots

Knots determine *how* and *where* the basis functions are defined. The knots in the example shown in Figure 14-7 are placed at 0,1,2,3,4,5,6, and 7. This series of numbers is called the *knot sequence*. A knot sequence is specified as a series of non-decreasing real numbers. There can be multiple knots, where two or more knots have the same value. The term *knot multiplicity* is used to refer to the number of knots that have the same value for any one location.

Increasing the knot multiplicity causes the curve to move closer to the corresponding control point. At a knot multiplicity equal to *order-1*, the curve passes through the control point. In general, there may be a discontinuity in the curve at the point where the knot multiplicity is equal to the order.

Figure 14-8 shows an example of the basis functions for a cubic B-Spline with a multiple interior knot. The knot multiplicities are listed below the u -axis and each basis function is numbered. A knot of multiplicity of 3 exists at $u=5$. The basis functions always sum to 1, so as u approaches 5, basis function 4, $B_4(u)$, is increasing in value, while the other basis functions are decreasing in value. At $u=5$, $B_4=1$ and all the other basis functions are zero.

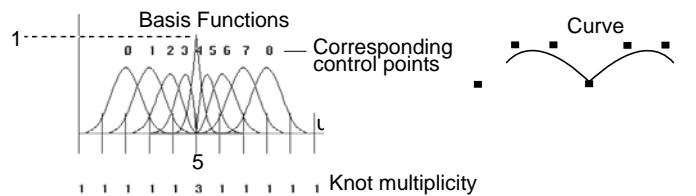


Figure 14-8 Basis Functions for a Cubic B-Spline with a Multiple Interior Knot

Figure 14-9 shows the relationships between the B-Spline and its basis functions, control points, and knots.

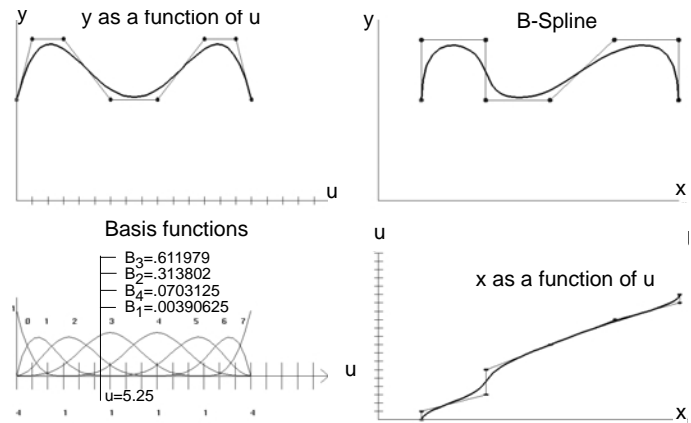


Figure 14-9 B-Spline Relationships

Review the rules governing B-Spline basis functions as you study the figure.

In this example of a cubic B-Spline, there are 8 control points and consequently there are 8 basis functions, which are numbered 0 through 7. (Rule 1)

The basis functions are positive (they attract the curve). (Rule 2)

The curve is defined where *order* number (4 in this case) of basis functions are defined. (Rule 3)

There are at most *order* number (4) of non-zero basis functions; for example, at $u=5.25$, the only non-zero basis functions are B_1 , B_2 , B_3 , and B_4 . (Rule 4)

Summing the basis functions ($B_1+B_2+B_3+B_4$) at $u=5.25$ gives:

$$.00390625 + .313802 + .611979 + .0703125 = 1 \text{ (Rule 5)}$$

The numbers below the u axis in the basis function plot represent the knot multiplicity. Adding up these numbers gives you the total number of knots. In this example, the number of knots (12) equals the number of control points (8) plus the order (4). (Rule 6)

The knot sequence (0,0,0,0,3,6,9,12,15,15,15,15) is non-decreasing. (Rule 7)

The B-Spline in Figure 14-9 has knot multiplicities equal to its order (4) at the first and last knots. This causes the B-Spline to interpolate (pass through) the first and last control points. By using this technique, you can make a B-Spline go through its endpoints, which is useful for joining B-Spline segments.

14.2.4 Weights

When you want to create a curve that is influenced more by a particular control point than by the others, you can assign a *weight* to alter that point's influence.

The polynomial curves that you have learned about so far have had equally weighted control points. When you increase the weight of a control point, you increase its influence on the curve. When the weight of one control point increases, the influence of the other nearby control points decreases. (Rule 5)

Figure 14-10 shows the same B-Spline as Figure 14-9 with increased weight at the third control point (d_2). This control point now exerts more influence on the curve. The greater influence can also be seen in the corresponding basis function (B_2). B_2 is now much larger than the other 3 basis functions. This means that d_2 now has more influence than its nearby control points. In the magnet analogy of control points, this is equivalent to having a strong magnet at d_2 and much weaker magnets of equal strengths at the other control points.

Note: Use positive values for weights.

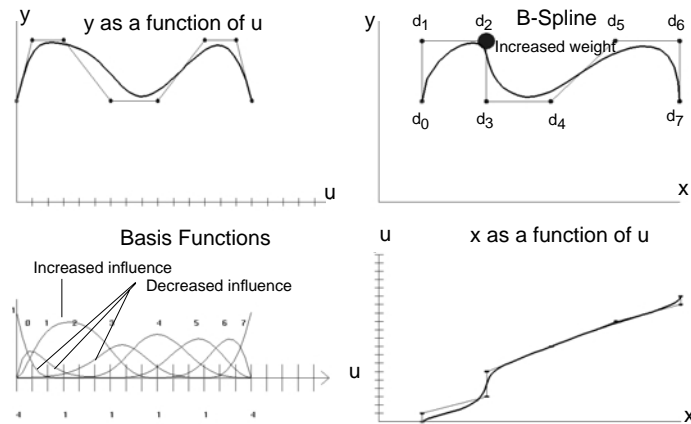


Figure 14-10 Increasing the Weight of a Control Point Increases Its Influence

14.3 GL NURBS Curves

So far, you have learned that curves can be represented with parametric polynomials. You know that polynomials can be joined to create a B-spline. You have seen how control points shape a B-spline and how knots and basis functions contribute to the definition of a B-Spline. You now have the information you need to understand what a NURBS representation is:

- Non-Uniform** Knots do not have to be spaced at equal intervals.
- Rational** Ratio of polynomials (see Note below) allows the weights of the control points to be specified.
- B-Spline** B-Spline, basis functions are used.

Note: Throughout this discussion, polynomial is used, rather than nonrational, to signify curves whose control points are equally weighted (have x,y,z coordinates, but no (or equal) w coordinates).

Use the `nurbcurve()` subroutine to draw GL NURBS curves, by specifying the knot sequence, the control points, and the order. As in other GL constructions, call the `nurbcurve()` subroutine between a `bgncurve()/endcurve()` pair

The ANSI C specification for `nurbcurve()` is:

```
void nurbcurve(long knot_count, double knot_list, long
offset, double *ctlarray, long order, long type);
```

where:

- knot_count* is the number of knots in *knot_list*
- knot_list* is an array of *knot_count* non-decreasing *knot_values*
- offset* is the offset in bytes between successive control points
- ctlarray* is an array of control points
- order* is the order of the curve (degree +1)
- type* is the type of curve, either `N_V3D` for a polynomial (x,y,z) curve, or `N_V3DR` for a rational (x,y,z,w) curve

Note: There are two other curve types, `N_P2D` and `N_P2DR`, used as trimming curves. See Section 14.5, "Trimming NURBS Surfaces."

Follow these rules for NURBS:

- $knot_count = \text{number of control points} + \text{order}$
- offset allows control points to be part of a structure
- control point coordinates are of form (x,y,z) for polynomial curves, (wx,wy,wz,w) for rational curves
- use positive weights
- maximum order is `getgdesc(GD_NURBS_ORDER)`

The following code fragment defines and draws a NURBS curve:

```
double ctlpts[4][3]={{-.75,-.75,0.0},{-.5,.75,0.0},{.5,.75,0.0},{.75,-.75,0.0}};
double knots[8] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
bgncurve();
    nurbscurve(8, knots, 3*sizeof(double), &ctlpts[0][0], 4, N_V3D);
endcurve();
```

NURBS have many interesting properties, for example, you can represent conic sections (circles, parabolas, ellipses, hyperbolas) exactly with NURBS. Figure 14-11 shows how to represent a circle with a quadratic NURBS curve:

```
double knots[14] = {0.0,0.0,0.0,2.5,2.5,5.,5.,7.5,7.5,10.,10.,10.};
double control_points[][4]={{9.,5.,0.1},{6.369,6.369,0.,.708},{5.,9.,0.,1.},
{.708,6.369,0.,.708}, {1.,5.,0.,1.}, {.708,.708,0.,.708}, {5.,1.,0.,1.},
{6.369,.708,0.,.708}, {9.,5.,0.,1.}};
```

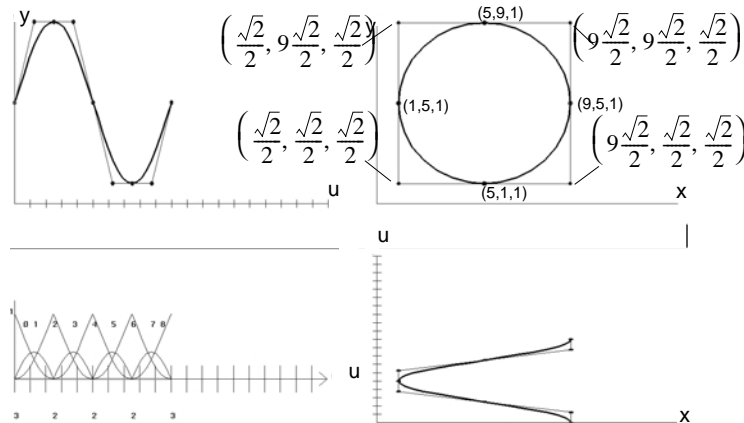


Figure 14-11 Representing a Circle with NURBS

14.4 NURBS Surfaces

A *parametric surface* is traced out as the two parameters u and v take on values in the domain (shaded region of the real plane), as shown in Figure 14-12. Parametric surfaces are analogous to the parametric curves you learned about in the beginning of the chapter, except that the domain now has two parameters, the basis functions are now three-dimensional, and the control points now connect to form a *control net*.

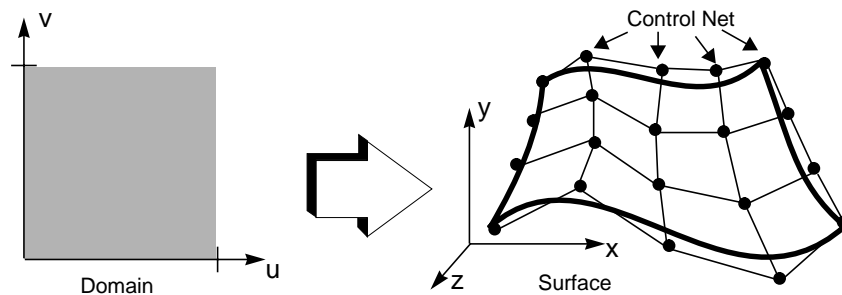


Figure 14-12 Parametric NURBS Surface

To create a NURBS surface, call `nurbssurface()` within a `begnsurface()/endsurface()` pair.

There are three types of NURBS surfaces in the GL:

- *Geometric surface* - defines the geometry of the surface
- *Color surface* - defines the color of the surface as a NURBS surface
- *Texture surface* - defines the texture of the surface as a NURBS surface

Define one (and only one) geometric surface within each `begn/endsurface` pair. You can optionally specify one color surface and one texture surface per geometric surface.

If you do specify a color and/or texture surface, the defined color/texture is applied to the NURBS surface in the same way that color-per-vertex or texture mapping is applied to polygons. Defining a color or texture NURBS surface is one way to apply color and texture to NURBS surfaces, but it is not the only way. See Chapter 4, “Display and Color Modes” and Chapter 18, “Texture” for more information about defining colors and textures.

You can also use the standard lighting models when rendering NURBS surfaces. With no other modifications, the NURBS surface appears in the currently bound material and with the currently bound lights and lighting model.

To draw a NURBS surface, specify:

- Control points - an array of data of the appropriate type for the surface
- Knots - a set of non-decreasing numbers for each parameter
- Order - (degree + 1) for each of the two surface parameters

There may be a different order and different knot sequence for each parameter.

As in the curve case, certain dependencies exist between the surface orders, the knot counts, and the number of control points. You specify the surface orders and the knot counts to determine the number and arrangement of the control points.

If O_u and O_v are the surface orders in the u and v directions, and if K_u and K_v are the knot counts in those directions, then the control points form an array of size: $(K_u - O_u) \times (K_v - O_v)$

The ANSI C specification for `nurbssurface` is:

```
void nurbssurface(long u_knot_count, double u_knots[],
                 long v_knot_count, double v_knots[],
                 long u_offset, long v_offset,
                 double *ctlarray,
                 long u_order, long v_order, long type);
```

The arguments to `nurbssurface()` are:

<i>u_knot_count</i>	number of knots in the u parameter of the domain
<i>u_knot</i>	array of <i>u_knot_count</i> nondecreasing knot values
<i>v_knot_count</i>	number of knots in the v parameter of the domain
<i>v_knot</i>	array of <i>v_knot_count</i> non-decreasing knot values
<i>u_offset</i>	offset in bytes between successive control points in <i>ctlarray</i> in the u parameter of the domain
<i>v_offset</i>	offset in bytes between successive control points in <i>ctlarray</i> in the v parameter of the domain

<code>ctlarray</code>	array of control points
<code>u_order</code>	order of the surface in u
<code>v_order</code>	order of the surface in v
<code>type</code>	type of surface, indicated by one of these types:
<code>N_V3D</code>	control points define a geometric surface in double-precision coordinates of the form (x, y, z) .
<code>N_V3DR</code>	control points define a geometric surface in double-precision homogeneous coordinates of the form (wx, wy, wz, w) .
<code>N_C4D</code>	control points define a color surface in double-precision color components of the form $(R, G, B, \text{and } A)$.
<code>N_C4DR</code>	control points define a color surface in double-precision color components that are in homogeneous form.
<code>N_T2D</code>	control points define a texture surface in double-precision texture coordinates that exist in a two-dimensional (s and t) texture space.
<code>N_T2DR</code>	control points define a texture surface in double-precision texture coordinates that are in homogeneous form.

Note: When specifying control points, the coordinates must be appropriate for *type*. That is, if you specify *type* `N_V3D` for the point geometry, *ctlarray* must contain elements in the form of (x, y, z) triples. If you specify `N_C4D`, *ctlarray* must contain an array of (R, G, B, A) components in double-precision form.

This interface is powerful in that the only requirement is that the x, y, z , and w coordinates of a control point are placed in successive memory locations. However, the control points themselves need not be contiguous. The data can be part of a larger data structure, or the points can form part of a larger array.

For example, suppose the data appears as follows:

```
struct ptdata {
    long tag1, tag2; double x, y, z, w;
}
points[4][4][3];
```

To use offsets, set *u_offset* to $4*3*\text{sizeof}(\text{double})$, *v_offset* to $3*\text{sizeof}(\text{double})$, and *ctlarray* to $\&\text{points}[0][0][0]$.

14.4.1 Geometric Surfaces

To draw a geometric surface, call `nurbssurface()` between a `bgnsurface()/endsurface()` pair and specify the surface type as `N_V3D` or `N_V3DR`.

For example, to define a geometric surface for which you have specified the knot sequence as *knots* and the control points as *ctlpoints*, use:

```
double ctlpoints[4][4][3];
double knots[8] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};

/*
Initializes the control points of the surface to a small hill.
The control points range from -3 to +3 in x, y, and z
*/
void init_surface(void)
{
    int u, v;

    for (u = 0; u < 4; u++)
    {
        for (v = 0; v < 4; v++)
        {
            ctlpoints[u][v][0] = 2.0*((double)u - 1.5);
            ctlpoints[u][v][1] = 2.0*((double)v - 1.5);

            if ( (u == 1 || u == 2) && (v == 1 || v == 2) )
                ctlpoints[u][v][2] = 3.0;
            else
                ctlpoints[u][v][2] = -3.0;
        }
    }
}

bgnsurface();
nurbssurface(
    8, knots,
    8, knots,
    4*3*sizeof(double), 3*sizeof(double),
    &ctlpoints[0][0][0],
    4, 4,
    N_V3D);
endsurface();
```

14.4.2 Color Surfaces

To specify a color surface, you use the same syntax as you do for a geometric surface, but you specify the type as `N_C4D` or `N_C4DR`. The color that you specify as a color surface has no effect when lighting is on (unless you have called `lmcolor()` with an appropriate argument), just as color commands have no effect on polygons when lighting is on. The color surface has no underlying geometry of its own; it is simply a property that is applied to the NURBS surface.

14.4.3 Texture Surfaces

To specify a texture surface, you use the same syntax as you do for a geometric surface, but you specify the type as `N_T2D` or `N_T2DR`. For texture mapping, the texture surface passed to the `nurbssurface()` call must contain an array of *s* and *t* texture coordinates that are associated with the texture surface.

The texture is applied to the NURBS surface, taking into account the current color defined by the current lighting model, in conjunction with the color defined by a color `nurbssurface()` call. You must call `texbind` to define a current texture for the texture coordinates to have any effect. For more information on defining textures, see Chapter 18, "Texture Mapping".

Note: For color and texture surfaces, the number, spacing, or sequence of the control points is completely independent of the number, spacing, or sequence of the control points that define the geometric surface.

14.5 Trimming NURBS Surfaces

You can *trim* a NURBS surface to define its visible regions. The trimming information is mapped to model space along with the surface, as shown in Figure 14-13. A *trimming loop* defines the trim regions (shown in white) in the domain (shown in gray). Trimming loops can be composed of one or more *trimming curves*.

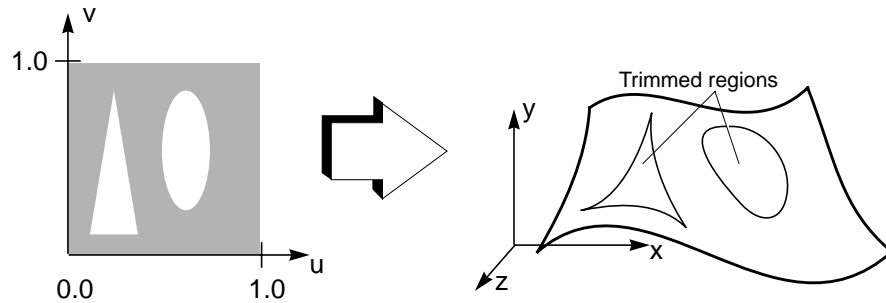


Figure 14-13 Trimming Surface

Trimming curves, also called edges, are defined in the domain of the NURBS surface. Each edge has a head and a tail. The edge runs in the direction indicated starting at its tail and ending at its head.

You can define trimming curves with NURBS curves, using `nurbcurve()`, or with piecewise linear curves (a series of line segments), using `pwlcurve()`, or any combination of the two.

A trimming loop is a closed, oriented curve, defined in the domain, composed of edges that meet head-to-tail. Trimming loops are specified within the `bgnsurface()/endsurface()` block that defines the geometric surface.

Use the orientation of the trimming curve to indicate which regions of the surface to display:

- If the edges of the trimming curve run clockwise, the region of the NURBS surface that lies inside of the trimming curve *is not* displayed.
- If the edges of the trimming curve run counterclockwise, the region of the NURBS surface that lies inside of the trimming curve *is* displayed.

You can define a trimming loop with a single NURBS curve, or with a piecewise linear curve, or with a series of curves (of either type) joined head-to-tail.

Figure 14-14 shows some trimming curves and loops. If you nest trimming loops, as shown in Figure 14-14, you must specify the outer-most loop as counterclockwise.

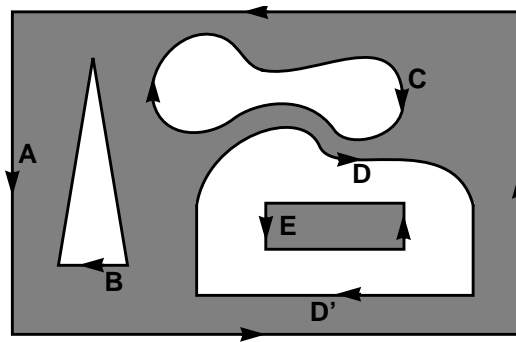


Figure 14-14 Trimming Curves

The following pseudocode creates the trimming curves shown in Figure 14-14:

```
bgnsurface();
  nurbsurface(...);
  bgntrim();
    pwlcurve(...); /* A */
  endtrim();
  bgntrim();
    pwlcurve(...); /* B */
  endtrim();
  bgntrim();
    nurbscurve(...); /* C */
  endtrim();
  bgntrim();
    nurbscurve(...); /* D */
    pwlcurve(...); /* D' */
  endtrim();
  bgntrim();
    pwlcurve(...); /* E */
  endtrim();
endsurface();
```

The following rules apply to trimming curves and trimming loops:

- Each trimming loop is surrounded by a `bgntrim()/endtrim()` pair.
- Each trimming loop must be closed; that is, the coordinates of the first and last points of the trimming curve must be identical (within a tolerance of 10^{-6}).
- Within a multi-segment trimming loop, the trimming curves must connect head-to-tail. This means that the last point of each curve segment must touch the first point of the next segment, and the last point of the last segment must touch the first point of the first segment.
- Trimming loops can neither touch nor intersect (except at their end points, which must touch).
- The outer loop of a nested trim sequence must be counter-clockwise.
- The trim space is the parameter space as defined by the knot sequence. For example, for a surface with identical *u* and *v* knot sequences (0,0,0,0,10,10,10,10), with no trimming information provided the trim region is effectively an isoparametric square with corners at (0,0) and (10,10).

To specify a NURBS trimming curve, use `nurbscurve()`, with `N_P2D` data for polynomial (*u,v*) curves or `N_P2DR` data for rational (*wu,wv,w*) curves.

To specify a piecewise linear trimming curve, use `pwlcurve()`:

```
void pwlcurve(long n, double *data, long offset, long type);
```

<i>n</i>	number of points in the trimming curve
<i>data</i>	array of points on the trimming curve
<i>offset</i>	offset in bytes between points in the array
<i>type</i>	type of curve, use <code>N_ST</code>

An offset allows the data points to be part of an array of larger structural elements. `pwlcurve()` searches for the *n*th coordinate pair beginning at `data_array + n * offset`.

The trim curve is drawn in the domain by connecting each point in the *data* array to the next point. If this `pwlcurve()` is the only curve forming a trimming loop, it is important to increment the trim point count as it is to duplicate the last point—in other words, although the last and first points are identical, they must be specified and counted twice.

14.6 NURBS Properties

NURBS properties control the rendering of NURBS curves and surfaces for the current window. You can set and query NURBS properties on a per-window basis. Use `setnurbsproperty()` to define the current NURBS properties. Use `getnurbsproperty()` to query the system for the current NURBS properties.

The ANSI C specifications for these two functions are:

```
void setnurbsproperty(long property, float value)
void getnurbsproperty(long property, float *value)
```

For maximum generality, express the value of a property in floating point. For some properties, only integer values make sense, but you must still pass them in floating point form.

Each property has a reasonable default value, but can be changed to affect the accuracy of some part of the rendering.

The NURBS properties are:

<code>N_PIXEL_TOLERANCE</code>	A positive floating point value that bounds the maximum length (in pixels) of an edge of a line segment generated in the <i>tessellation</i> (breaking down into triangles) of a curve or a polygon generated in the tessellation of a surface.
<code>N_ERRORCHECKING</code>	A Boolean value that, when TRUE, instructs the GL to send NURBS-related error messages to standard error.
<code>N_DISPLAY</code>	An enumeration that dictates the surface rendering format; it affects surfaces only. Possible values are <code>N_FILL</code> , <code>N_OUTLINE_POLY</code> , and <code>N_OUTLINE_PATCH</code> . <code>N_FILL</code> instructs the GL to fill all polygons generated in the tessellation of the surface. <code>N_OUTLINE_POLY</code> instructs the GL to outline all polygons generated. <code>N_OUTLINE_PATCH</code> instructs the GL to outline the boundary of all surface patches and trim curves.
<code>N_CULLING</code>	A Boolean value that, when TRUE, instructs the GL to discard before tessellation all patches that are outside the current viewport.

14.7 Sample NURBS Program

This sample program, *nurbs.c*, draws a lighted nurbs surface. Use the mouse to toggle the trimming curve on and off.

```
/*
 * nurbs.c
 */

#include <stdio.h>
#include <gl/gl.h>
#include <gl/device.h>

#define S_NUMKNOTS8 /* number of knots in each dimension of surface*/
#define S_NUMCOORDS3 /* number of surface coordinates, (x,y,z) */
#define S_ORDER4 /* surface is bicubic, order 4 for each parameter */
#define T_NUMKNOTS12 /* number of knots in the trimming curve */
#define T_NUMCOORDS3 /* number of curve coordinates, (wx,wy,w) */
#define T_ORDER3 /* trimming curve is rational quadratic */

/* number of control points in each dimension of NURBS */
#define S_NUMPOINTS(S_NUMKNOTS - S_ORDER)
#define T_NUMPOINTS(T_NUMKNOTS - T_ORDER)
/* trimming */
int trim_flag = 0;

long zfar;

double surfknots[S_NUMKNOTS] = {
    -1., -1., -1., -1., 1., 1., 1., 1.
};

double ctlpoints[S_NUMPOINTS][S_NUMPOINTS * S_NUMCOORDS] = {
    -2.5, -3.7, 1.0,
    -1.5, -3.7, 3.0,
    1.5, -3.7, -2.5,
    2.5, -3.7, -.75,

    -2.5, -2.0, 3.0,
    -1.5, -2.0, 4.0,
    1.5, -2.0, -3.0,
    2.5, -2.0, 0.0,

    -2.5, 2.0, 1.0,
    -1.5, 2.0, 0.0,
    1.5, 2.0, -1.0,
```



```

        2.5, 2.0, 2.0,

-2.5, 2.7, 1.25,
-1.5, 2.7, .1,
  1.5, 2.7, -.6,
  2.5, 2.7, .2
};

double trimknots[T_NUMKNOTS] = {
    0., 0., 0., 1., 1., 2., 2., 3., 3., 4., 4., 4.
};

double trimpoints[T_NUMPOINTS][T_NUMCOORDS] = {
    1.0, 0.0, 1.0,
    1.0, 1.0, 1.0,
    0.0, 2.0, 2.0,
    -1.0, 1.0, 1.0,
    -1.0, 0.0, 1.0,
    -1.0, -1.0, 1.0,
    0.0, -2.0, 2.0,
    1.0, -1.0, 1.0,
    1.0, 0.0, 1.0,
};

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()
{
    long dev;
    short val;

    init_windows();
    setup_queue();
    init_view();
    make_lights();

    set_scene();
    setnurbsproperty( N_ERRORCHECKING, 1.0 );
    setnurbsproperty( N_PIXEL_TOLERANCE, 50.0 );
    draw_trim_surface();
}

```

```

while(TRUE) {
    while(qtest()) {
        dev=qread(&val);
        switch(dev) {
            case ESCKEY:
                gexit();
                exit(0);
                break;
            case WINQUIT:
                gexit();
                dglclose(-1); /* this for DGL only */
                exit(0);
                break;
            case REDRAW:
                reshapeviewport();
                set_scene();
                draw_trim_surface();
                break;
            case LEFTMOUSE:
                if( val )
                    trim_flag = !trim_flag; /* trimming */
                break;
            default:
                break;
        }
    }
    set_scene();
    draw_trim_surface();
}

init_windows()
/*-----
 * Initialize all windows
 *-----
*/
{
    if (getgdesc(GD_BITS_NORM_DBL_RED) <= 0) {
        fprintf(stderr, "nurbs: requires double buffered RGB which is "
            "unavailable on this machine\n");
        exit(1);
        /* NOTREACHED */
    }
    winopen("nurbs");
    wintitle("NURBS Surface");
    doublebuffer();
}

```

```

    RGBmode();
    gconfig();
    zbuffer( TRUE );
    glcompat(GLC_ZRANGEMAP, 0);
    zfar = getgdesc(GD_ZMAX);
}

setup_queue()
/*-----
 * Queue all devices
 *-----
 */
{
    qdevice(ESCKEY);
    qdevice(RIGHTMOUSE);
    qdevice(WINQUIT);
    qdevice(LEFTMOUSE);/* trimming */
}

init_view()
/*-----
 * Initialize view and lighting mode
 *-----
 */
{
    mmode(MPROJECTION);
    ortho( -4., 4., -4., 4., -4., 4. );

    mmode(MVIEWING);
    loadmatrix(idmat);
}

set_scene()
/*-----
 * Clear screen and rotate object
 *-----
 */
{
    lmbind(MATERIAL, 0);
    RGBcolor(150,150,150);
    lmbind(MATERIAL, 1);

    czclear(0x00969696, zfar);

    rotate( 100, 'y' );
}

```

```

rotate( 100, 'z' );
}

draw_trim_surface()
/*-----
 * Draw NURBS surface
 *-----
 */
{
  bgnsurface();
  nurbsurface(
    sizeof( surfknots ) / sizeof( double ), surfknots,
    sizeof( surfknots ) / sizeof( double ), surfknots,
    sizeof(double) * S_NUMCOORDS,
    sizeof(double) * S_NUMPOINTS * S_NUMCOORDS,
    ctlpoints,
    S_ORDER, S_ORDER,
    N_V3D
  );
  /* trimming */
  if( trim_flag ) {
    bntrim();
    /* trim curve is a rational quadratic nurbscurve (circle) */
    nurbscurve(
      sizeof( trimknots ) / sizeof( double ),
      trimknots,
      sizeof(double) * T_NUMCOORDS,
      trimpoints,
      T_ORDER,
      N_P2DR
    );
    endtrim();
  }
  endsurface();
  swapbuffers();
}

make_lights()
/*-----
 * Define material, light, and model
 *-----
 */
{
  /* initialize model and light to default */
  lndef(DEFLMODEL,1,0,0);
  lndef(DEFLIGHT,1,0,0);
}

```

```

/* define material #1 */
{
static float array[] = {
    EMISSION, 0.0, 0.0, 0.0,
    AMBIENT, 0.1, 0.1, 0.1,
    DIFFUSE, 0.6, 0.3, 0.3,
    SPECULAR, 0.0, 0.6, 0.0,
    SHININESS, 2.0,
    LNULL
};
lmdf(DEFMATERIAL, 1, sizeof(array)/sizeof(array[0]), array);
}

/* turn on lighting */
lmbind(LIGHT0, 1);
lmbind(LMODEL, 1);
lmbind(MATERIAL, 1);
}

```

Suggestions for Further Reading

For a basic introduction to curves and surfaces:

Foley, J.D., A. van Dam, S. Feiner, and J.D. Hughes, "Representing Curves and Surfaces," in *Computer Graphics Principles and Practice*, 2d ed., Addison Wesley Publishing Company Inc., Menlo Park, 1990.

For a comprehensive treatment, including derivations:

Farin, G., *Curves and Surfaces for Computer Aided Geometric Design*, Academic Press, 2d ed., New York, 1990.

14.8 Old-Style Curves and Surfaces

This section describes the methods used for drawing curves and surfaces prior to the 4D1-3.2 release of the Graphics Library software. These techniques are still supported for compatibility with programs written for earlier versions of the software.

Note: Use of these statements is not recommended. All new development should use the GL NURBS method.

The techniques presented in this section have limited capabilities. Old-style curves are restricted to *cubic splines* (degree 3, order 4) only. Each polynomial segment is presented to the drawing subroutines one at a time.

Old-style surfaces are restricted to *bicubic wireframe* surfaces only, and each surface patch is presented to the drawing subroutines one at a time.

14.8.1 Old-Style Curves

The curves in most applications are too complex to be represented by a single curve segment and instead must be represented by a series of curve segments joined end-to-end. To create smooth joints, you must control the positions and curvatures at the endpoints of curve segments.

The shape of the curve segment is determined by a function of a set of four control points. The IRIS approximates the shape of a curve segment with a series of straight line segments.

A set of constraints expresses how the shape of the curve segment relates to the control points. For example, a constraint might be that one endpoint of the curve segment is located at the first control point, or that the tangent vector at an endpoint lies on the line segment formed by the first two control points. This constraint information is used to define a *basis matrix* for the curve.

The old-style IRIS curve facility is based on the *parametric cubic curve*. Three classes of cubic curves are available:

- Bezier
- Cardinal spline
- B-spline

The characteristics of each of these curves are summarized next.

Bezier Cubic Curve

A *Bezier* cubic curve segment passes through the first and fourth control points and uses the second and third points to determine the shape of the curve segment. Of the three kinds of curves, the Bezier form provides the most intuitive control over the shape of the curve.

Cardinal Spline Cubic Curve

A *Cardinal spline* curve segment passes through the two interior control points and is continuous in the first derivative at the points where segments meet. The curve segment starts at the second point and ends at the third point, and uses the first and fourth points to define the shape of the curve. The mathematical derivation of the Cardinal spline basis matrix can be found in James H. Clark, *Parametric Curves, Surfaces and Volumes in Computer Graphics and Computer-Aided Geometric Design*, Technical Report No. 221, Computer Systems Laboratory, Stanford University.

B-Spline Cubic Curve

In general, a *B-spline* curve segment does not pass through any control points, but is continuous in both the first and second derivatives at the points where segments meet. Thus, a series of joined B-spline curve segments is smoother than a series of Cardinal spline segments.

14.8.2 Drawing Old-Style Curves

You can create complex curved lines by joining several curve segments end-to-end. The curve facility provides the means for making smooth joints between the segments.

You draw an old-style curve segment by specifying:

- a set of four control points
- a basis, which defines how the system uses the control points to determine the shape of the segment

To draw an old-style curve segment:

1. Define and name a basis matrix with `defbasis()`:

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

The matrix *mat* is saved and is associated with the identifier *id*. Use *id* in subsequent calls to `curvebasis()` and `patchbasis()`.

2. Select a defined basis matrix (defined by `defbasis()`) as the current basis matrix with `curvebasis()`:

```
void curvebasis(short basisid);
```

3. Specify the number of line segments used to approximate each curve segment with `curveprecision()`:

```
void curveprecision(short nsegments);
```

When `crv()`, `crvn()`, `rcrv()`, or `rcrvn()` executes, a number of straight line segments (*nsegments*) approximates each curve segment. The greater the value of *nsegments*, the smoother the curve, but the longer the drawing time.

4. Draw the curve segment using the current basis matrix, the current curve precision, and the four control points with `crv()` (`rcrv()` draws a rational curve).

```
void crv(Coord geom[4][3]);
```

When you call `crv()`, a matrix *M* is built from the geometry *G* (the control point array), the current basis M_b , and the current precision *n*, as shown below:

$$\begin{bmatrix} \frac{6}{n^3} & 0 & 0 & 0 \\ \frac{6}{n^3} & \frac{2}{n^2} & 0 & 0 \\ \frac{1}{n^3} & \frac{1}{n^2} & \frac{1}{n} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} M_b G$$

The bottom row of the resulting transformation matrix identifies the first of *n* points that describe the curve.

A forward difference algorithm is used to iterate the matrix to generate the remaining points in the curve. Each iteration draws one line segment of the curve segment. At each iteration, the first row is added to the second row, the second row is added to the third row, and the third row is added to the fourth row. The fourth row is then output as one of the points on the curve.

```

/* This is the forward difference algorithm */
/* M is the current transformation matrix */
move (M[3][0]/M[3][3], M[3][1]/M[3][3], M[3][2]/M[3][3]);
/* iteration loop */
for (cnt = 0; cnt < iterationcount; cnt++) {
    for (i=3; i>0; i--)
        for (j=0; j<4; j++)
            [j] = M[i][j] + M[i-1][j];
    draw(M[3][0]/M[3][3], M[3][1]/M[3][3], M[3][2]/M[3][3]);
}

```

Each iteration draws one line segment of the curve. For the precision matrix shown, these points are generated:

$$(0, 0, 0, 1), \left(\left(\frac{1}{n}\right)^3, \left(\frac{2}{n}\right)^2, \frac{1}{n}, 1\right), \left(\left(\frac{2}{n}\right)^3, \left(\frac{2}{n}\right)^2, \frac{2}{n}, 1\right), \left(\left(\frac{3}{n}\right)^3, \left(\frac{3}{n}\right)^2, \frac{3}{n}, 1\right), \dots$$

The iteration loop of the forward difference algorithm is implemented in the Geometry Pipeline. You can use `curveit()` to access the forward difference algorithm, making it possible to generate a curve directly from a forward difference matrix:

```
void curveit(short iterationcount);
```

`curveit()` iterates the current matrix (the one on top of the matrix stack) *iterationcount* times. Each iteration draws one of the line segments that approximate the curve. `curveit()` does not execute the initial move in the forward difference algorithm. A `move(0.0, 0.0, 0.0)` call must precede `curveit()` so that the correct first point is generated from the forward difference matrix.

Note: To get the numbers to use for the basis matrix M_b , see the sample program *patch1.c* at the end of the next section.

Drawing a Series of Curve Segments

You use `crvn()` to draw a series of cubic spline segments using the current basis, precision, and a series of n control points. Calling `crvn()` has the same effect as programming a sequence of `crv()` calls with overlapping control points.

```
void crvn(long n, Coord geom[][3]);
```

The control points specified in `geom` determine the shapes of the curve segments and are used four at a time. If the current basis is a B-spline, Cardinal spline, or a basis with similar properties to these types of curves, the curve segments are joined end-to-end and appear as a single curve.

When you call `crvn()` with a Cardinal spline or B-spline basis, it produces a single curve. However, calling `crvn()` with a Bezier basis produces several separate curve segments. As with `crv()` and `rcrv()`, a precision and basis must be defined before calling `crvn()` or `rcrvn()`. This is true even if the routines are compiled into objects. See Chapter 16 for more information on graphical objects.

Rational Curves

The IRIS graphics hardware actually works in homogeneous coordinates x , y , z , and w , where 3-D coordinates are given by xw , yw , and zw . The homogeneous character of the system is not obvious because w is normally the constant 1.

The w coordinate can also be expressed as a cubic function of the parameter t , so that the 3-D coordinates of points along the curve are given as a quotient of two cubic polynomials. The only constraint is that the denominator for all three coordinates must be the same. When w is not the constant 1, but some cubic polynomial function of t , the curves generated are usually called *rational cubic splines*.

The basis definitions for rational cubic splines are identical to those for cubic splines, as are the precision specifications. The only difference is that the geometry matrix must be specified in four-dimensional homogeneous coordinates.

Use `rcrv()` to draw rational curves:

```
void rcrv(Coord geom[4][4]);
```

`rcrv()` draws a rational curve segment using the current basis matrix, the current curve precision, and the four control points specified in its argument. `rcrv()` is analogous to `crv()` except that w coordinates are included in the control point definitions.

`rcrvn()` takes a series of n control points and draws a series of rational cubic spline curve segments using the current basis and precision:

```
void rcrvn(long n, Coord geom[][4]);
```

The control points specified in *geom* determine the shapes of the curve segments and are used four at a time.

14.8.3 Drawing Old-Style Surfaces

A *surface patch* appears on the screen as a wireframe of curve segments. A set of user-defined control points determines the shape of the patch. You can create complex surfaces by joining several patches into one large patch. You can also create a complex surface consisting of several joined patches by using overlapping sets of control points and the B-spline and Cardinal spline curve bases.

The method for drawing old-style surfaces is similar to that for drawing curves. You draw an old-style wireframe surface patch by specifying:

- a set of 16 control points
- the number of curve segments to be drawn in each direction of the patch
- the two bases that define how the control points determine the shape of the patch

The IRIS old-style surface facility is based on 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 patch are defined by varying the parameters u and v from 0 to 1. If one parameter is held constant and the other is varied from 0 to 1, the result is a cubic curve. Thus, you can create a wireframe patch by holding u constant at several values and using the IRIS curve facility to draw curve segments in one direction, and then doing the same for v in the other direction.

To draw a surface patch:

1. Define the appropriate curve bases using `defbasis()`. 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. Specify a basis matrix (defined by `defbasis()`) for the u and v parametric directions of a surface patch with `patchbasis()`:

```
void patchbasis(long uid, long vid);
```

The u basis and the v basis do not have to be the same. `patch()` uses the current u and v when it executes.

3. Specify the number of curve segments to be drawn in each direction with `patchcurves`:

```
void patchcurves(long ucurves, long vcurves);
```

A different number of curve segments can be drawn in each direction.

4. Specify the precisions for the curve segments in each direction with `patchprecision()`:

```
void patchprecision(long usegments, long vsegments);
```

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 that form the wireframe actually intersect.

5. Draw the surfaces with `patch` or `rpatch()`:

```
void patch(Matrix geomx, Matrix geomy, Matrix geomz);  
void rpatch(Matrix geomx, Matrix geomy, Matrix geomz, Matrix geomw);
```

`rpatch()` is the same as `patch`, except it draws a rational surface patch. The curve segments in the patch are drawn using the current linestyle, linewidth, color, and writemask.

The arguments `geomx`, `geomy`, and `geomz` are 4x4 matrices containing the coordinates of the 16 control points that determine the shape of the patch. `geomw` specifies the rational component of the patch to `rpatch()`.

The sample program *patch1.c* shows the number to use for each type of basis matrix discussed and it uses them to draw a surface patch.

```
#include <gl/gl.h>

#define X      0
#define Y      1
#define Z      2
#define XYZ    3

#define BEZIER  1
#define CARDINAL 2
#define BSPLINE 3

Matrix beziermatrix = {
    {-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}
};

Matrix cardinalmatrix = {
    {-0.5, 1.5, -1.5, 0.5},
    { 1.0, -2.5, 2.0, -0.5},
    {-0.5, 0.0, 0.5, 0.0},
    { 0.0, 1.0, 0.0, 0.0}
};

Matrix bsplinematrix = {
    {-1.0/6.0, 3.0/6.0, -3.0/6.0, 1.0/6.0},
    { 3.0/6.0, -6.0/6.0, 3.0/6.0, 0.0},
    {-3.0/6.0, 0.0, 3.0/6.0, 0.0},
    { 1.0/6.0, 4.0/6.0, 1.0/6.0, 0.0}
};

Matrix geom[XYZ] = {
    { { 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} },
    { {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} },
    { {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} },
    { {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} }
};
```

```

        { {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()
{
    int i, j;

    prefsiz(400, 400);
    winopen("patch1");
    color(BLACK);
    clear();
    ortho(-100.0, 800.0, -100.0, 800.0, -800.0, 100.0);

    /* define 3 types of bases */
    defbasis(BEZIER, beziermatrix);
    defbasis(CARDINAL, cardinalmatrix);
    defbasis(BSPLINE, bsplinematrix);

    /*
     * seven curve segments will be drawn in the u direction
     * and four in the v direction
     */
    patchcurves(4, 7);

    /*
     * the curve segments in u direction will consist of 20 line
     segments
     * (the lowest multiple of vcurves greater than usegments) and
     the curve
     * segments in the v direction will consist of 21 line
     segments (the
     * lowest multiple of ucurves greater than vsegments)
     */
    patchprecision(20, 20);

    /* the patch is drawn based on the sixteen specified control
    points */
    patchbasis(BEZIER, BEZIER);
    color(RED);
    patch(geom[X], geom[Y], geom[Z]);

    /*

```

```

    * another patch is drawn using the same control points but a
different
    * basis
    */
    patchbasis(CARDINAL, CARDINAL);
    color(GREEN);
    patch(geom[X], geom[Y], geom[Z]);

    /* a third patch is drawn */
    patchbasis(BSPLINE, BSPLINE);
    color(BLUE);
    patch(geom[X], geom[Y], geom[Z]);

    /* show the control points */
    color(WHITE);
    for (i = 0; i < 4; i++) {
        for (j = 0; j < 4; j++) {
            pushmatrix();
            translate(geom[X][i][j], geom[Y][i][j],
geom[Z][i][j]);
            circf(0.0, 0.0, 3.0);
            popmatrix();
        }
    }

    sleep(10);
    gexit();
    return 0;
}

```


Chapter 15

Antialiasing

This chapter describes methods for reducing display artifacts.

- Section 15.1, “Sampling Accuracy,” describes how to smooth scan conversion by enabling primitives to be drawn offset from pixel centers.
- Section 15.3, “One-Pass Antialiasing—the Smooth Primitives,” describes methods for antialiasing RGB images using a pixel-filling technique.
- Section 15.4, “Multipass Antialiasing with the Accumulation Buffer,” describes techniques for quickly generating antialiased points, lines, and polygons.
- Section 15.5, “Antialiasing on RealityEngine Systems,” describes the advanced multiple sampling feature of RealityEngine systems.

15.1 Sampling Accuracy

You may have noticed that lines displayed on a monitor appear to be made of a stairstep series of dots that make the line look jagged. Lines appear jagged because the true mathematical line is approximated by a series of points that are forced to lie on the pixel grid. Except in a few special cases (horizontal, vertical, and 45-degree lines) many of the approximating pixels are not on the mathematical line. Near-horizontal and near-vertical lines appear especially jagged, because their pixel approximations are a sequence of exactly horizontal or vertical segments, each offset one pixel from the next.

The jaggedness that you see is called *aliasing*, and techniques to eliminate or reduce aliasing are called *antialiasing*.

The following program, *jagged.c*, illustrates the aliasing problem.

```
/* Drag a color map aliased line with the cursor.*/

#include <gl/gl.h>
#include <gl/device.h>

#define WINSIZE 400

Device devs[2] = {MOUSEX,MOUSEY};
float orig[2] = {100.,100.};

main()
{
    short val, vals[2];
    long xorg, yorg;
    float vert[2];

    prefsiz(WINSIZE, WINSIZE);
    winopen("jagged");
    mmode(MVIEWING);
    ortho2(-0.5, WINSIZE-0.5, -0.5, WINSIZE-0.5);
    doublebuffer();
    gconfig();
    qdevice(ESCKEY);
    getorigin(&xorg, &yorg);

    while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
        color(BLACK);
        clear();
        getdev(2,devs,vals);
        vert[0] = vals[0] - xorg;
        vert[1] = vals[1] - yorg;
        color(WHITE);
        bgnline();
            v2f(orig);
            v2f(vert);
        endline();
        swapbuffers();
    }
    gexit();
    return 0;
}
```

This example draws a line from the point (100,100) to the current cursor position. Move the cursor around, and notice how jagged the line appears,

especially when it is nearly horizontal or nearly vertical. Even at angles far from vertical or horizontal there is some jaggedness, although it is not as noticeable.

The line you see on the screen is aliased because it is composed of discrete pixels, each set either to the color of the line or to the background color. A much better approximation can be developed by considering the exact mathematical line to be the center line of a rectangle of width one, extending the full length of the line. A correct, antialiased sampling of this rectangle onto the pixel grid takes into account the fraction of each pixel that is obscured by the rectangle, rather than simply selecting the pixels most obscured by it. Each pixel is then set to a color between the color of the line and the color of the background, based on the fraction of the pixel that is obscured, or covered, by the line's rectangle.

To correctly sample a point, line, or polygon, the fraction of every pixel covered by the exact projection of the primitive must be computed, and that fraction must be used to determine the color of the resulting pixel. Because mathematical points and lines have no area, and therefore cannot be sampled, it is necessary to define a geometry to be used for their sampling. Points are best thought of as circles of diameter one, centered around the exact mathematical point. Lines are rectangles of width one, centered around the exact mathematical line.

Figure 15-1 shows an antialiased line.

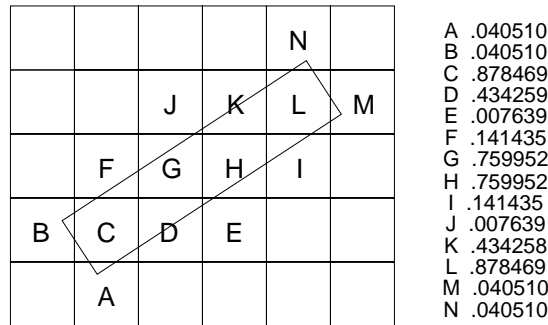


Figure 15-1 Antialiased Line

15.1.1 Subpixel Positioning

Vertices, after they have been transformed and projected to screen coordinates, are rounded to the nearest pixel center, rather than being treated with full precision. A prerequisite for accurate scan conversion of points, lines, and polygons is ensuring that their vertices are projected to the screen with *subpixel* precision. When you enable `subpixel()` mode, you defeat the default behavior of rounding projected vertices to the nearest pixel center. Exact vertex position is made available to the sampling hardware.

Use `subpixel()` to control the placement of point, line, and polygon vertices in screen coordinates:

```
void subpixel(Boolean b)
```

When `subpixel()` is FALSE, vertices are *snapped* (aligned) to the center of the nearest pixel after they have been transformed to screen coordinates. When `subpixel()` is TRUE, vertices are positioned with fractional precision.

The default setting for `subpixel()` depends on the system type. On IRIS Indigo Entry, XS, XS24, and Elan systems, polygons are always drawn with MicroPixel™ subpixel accuracy, regardless of the setting of `subpixel()`. On VGX, VGXT, SkyWriter, and RealityEngine systems, the default for `subpixel()` is FALSE, but performance is enhanced when subpixel positioning is enabled. It is thus a good idea to call `subpixel(TRUE)` when writing GL applications for these systems.

15.1.2 How Subpixel Positioning Affects Lines

In addition to its effect on vertex position, `subpixel()` also modifies the scan conversion of lines. Specifically, non-subpixel-positioned lines are drawn *closed*, meaning that connected line segments both draw the pixel at their shared vertex. subpixel-positioned lines are drawn *half open*, meaning that connected line segments share no pixels. Thus subpixel-positioned lines produce better results when `logicop` or `blendfunction()` are used, but will produce different, possibly undesirable results in 2-D applications where the endpoints of lines have been carefully placed.

For example, using the standard 2-D projection shown below, subpixel-positioned lines match non-subpixel-positioned lines pixel for pixel, except that they omit either the right-most or top-most pixel.

```
ortho2(left-0.5, right+0.5, bottom-0.5,top+0.5);
viewport (left,right,bottom,top);
```

Thus the non-subpixel-positioned line drawn from (0,0) to (0,2) fills pixels (0,0), (0,1), and (0,2), while the subpixel-positioned line drawn between the same coordinates fills only pixels (0,0) and (0,1).

On IRIS Indigo systems, subpixel-positioned lines are drawn closed rather than half-open. `subpixel()` is not supported by all models for all primitives. Refer to the `subpixel()` man page for details.

15.2 Blending

The pixel color value in the frame buffer is replaced with the incoming pixel color when pixels are drawn. When operating in RGB mode, however, it is possible to replace the color components of the frame buffer (destination) pixel with values that are a function of both the incoming (source) pixel color components and of the current frame buffer color components. This operation is called *blending*. Not all systems support blending. See the *blendfunction(3G)* man page for details.

The antialiasing techniques described in Section 15.3 require blending when operating in RGB mode. Blending has other uses, including drawing transparent objects, and compositing images. Blending is specified with the `blendfunction()` command:

```
void blendfunction (long sfactor,long dfactor);
```

The `blendfunction()` arguments *sfactor* and *dfactor* specify how destination pixels are computed, based on the incoming (source) pixel values and the current framebuffer values. The token specified for *sfactor* selects the blending factor used to scale the contribution from the source RGBA values. The token specified for *dfactor* selects the blending factor used to scale the contribution from the destination RGBA values.

Note: RealityEngine systems provide the ability to specify a constant color for blending. Use `blendcolor()` to set the values of the color components for the blending functions `BF_CA`, `BF_MCA`, `BF_CC`, and `BF_MCC`.

Blending factors *sfactr* and *dfactr* are chosen from the list of tokens in Table 15-1.

Token	Calculated Value	Notes
BF_ZERO	0.0	
BF_ONE	1.0	
BF_SA	Source Alpha / 255	
BF_MSA	1.0 – (source Alpha / 255)	
BF_DA	Source Alpha / 255	Requires alpha bitplanes
BF_MDA	1.0 – (source Alpha / 255)	Requires alpha bitplanes
BF_SC	Source RGBA / 255	<i>dfactr</i> only
BF_MSC	1.0 – (source RGBA / 255)	<i>dfactr</i> only
BF_DC	Destination RGBA / 255	<i>sfactr</i> only
BF_MDC	1.0 – (destination RGBA / 255)	<i>sfactr</i> only
BF_MIN_SA_MDA	Min (BF_SA, BF_MDA)	Requires alpha planes, changes expression
BF_CC	Constant RGBA / 255	RealityEngine only
BF_MCC	1 - (constant RGBA / 255)	RealityEngine only
BF_CA	Constant alpha / 255	RealityEngine only
BF_MCA	1 - (constant alpha / 255)	RealityEngine only
BF_MIN	Min(1, destination RGBA / source RGBA)	RealityEngine only
BF_MAX	Max(1, destination RGBA / source RGBA)	RealityEngine only

Table 15-1 Blending Factors

All the blending factors are defined to have values between 0.0 and 1.0 inclusive. In most cases, this range is obtained by dividing a color component value, in the range 0 through 255, by 255. The blending arithmetic is done such that a blending factor with a value of 1.0 has no effect on the color component that it weights. Also, a blending factor of 0.0 forces its corresponding color component to 0. Because the weighting factors are all positive, and because each weighted sum is clamped to 255, colors blend toward white, rather than wrapping back to low-intensity values.

Each frame buffer color component is replaced by a weighted sum of the current value and the incoming pixel value. This sum is clamped to a maximum of 255.

By default, *sfactor* is set to `BF_ONE` and *dfactor* is set to `BF_ZERO`, resulting in simple replacement of the framebuffer color components:

```
Rdestination = Rsource
Gdestination = Gsource
Bdestination = Bsource
Adestination = Asource
```

When `blendfunction()` is called with *sfactor* set to a value other than `BF_ONE`, or *dfactor* set to a value other than `BF_ZERO`, a more complex expression defines the frame buffer replacement algorithm.

```
Rdest = min (255, ((Rsource * sfactor) + (Rdest * dfactor)))
Gdest = min (255, ((Gsource * sfactor) + (Gdest * dfactor)))
Bdest = min (255, ((Bsource * sfactor) + (Bdest * dfactor)))
Adest = min (255, ((Asource * sfactor) + (Adest * dfactor)))
```

Blending factors `BF_DA`, `BF_MDA`, and `BF_MIN_SA_MDA` require alpha bitplanes for correct operation. Blending functions specified without using these three symbolic constants work correctly, regardless of the availability of alpha bitplanes.

Use the following command, testing for a nonzero return value, to determine if your machine has alpha bitplanes:

```
getgdesc(GD_BITS_NORM_SNG_ALPHA)
```

Blending factors `BF_SC`, `BF_MSC`, `BF_DC`, and `MF_MDC` weight each color component by the corresponding weight component. For example, you can scale each framebuffer color component by the incoming color component with the blending function:

```
blendfunction (BF_DC,BF_ZERO)
```

```
Rdestination = min (255, (Rsource * (Rdestination / 255)))
Gdestination = min (255, (Gsource * (Gdestination / 255)))
Bdestination = min (255, (Bsource * (Bdestination / 255)))
Adestination = min (255, (Asource * (Adestination / 255)))
```

The special blending factor `BF_MIN_SA_MDA` is intended to support polygon antialiasing, as described in Section 15.3.3. It must be used only for *sfactor*, and only while *dfactor* is `BF_ONE`. In this case, the blending equations are:

```
blendfunction (BF_MIN_SA_MDA,BF_ONE)
```

```
Rdestination = min (255, ((Rsource * sfactor) + Rdestination))
Gdestination = min (255, ((Gsource * sfactor) + Gdestination))
Bdestination = min (255, ((Bsource * sfactor) + Bdestination))
sfactor = min ((Asource/255), (1.0 - (Adestination/255)))
Adestination = sfactor + Adestination
```

This special blending function accumulates pixel contributions until the pixel is fully specified, then allows no further changes. Frame buffer alpha bitplanes, which must be present, store the accumulated contribution percentage, or “coverage”.

Although many blending functions are supported, the following function stands out as the single most useful one.

```
blendfunction (BF_SA,BF_MSA)
```

It weights incoming color components by the incoming alpha value, and frame buffer components by one minus the incoming alpha value. In other words, it blends between the incoming color and the frame buffer color, as a function of the incoming alpha. This function renders transparent objects by drawing them correctly when they are drawn in sorted order from farthest to nearest, specifying opacity as incoming alpha.

This sample program, *blendcircs.c*, illustrates image composition. Three colored circles are blended such that the first one is weighted by 0.5, the second by 0.35, and the third by 0.15. The blending function `blendfunction` (`BF_SA, BF_ONE`) is used, causing the colors to be added to each other, rather than blended. The order in which the circles are drawn makes no difference. Because the three weights add up to exactly 1.0, no clamping is done.

```
#include <stdio.h>
#include <gl/gl.h>
#define WINSIZE 400
#define RGB_BLACK 0x000000
#define RGB_RED 0x0000ff
#define RGB_GREEN 0x00ff00
#define RGB_BLUE 0xff0000
main()
{
    if (getgdesc(GD_BITS_NORM_SNG_RED) == 0) {
        fprintf(stderr, "Single buffered RGB not available\n");
        return 1;
    }
    if (getgdesc(GD_BLEND) == 0) {
        fprintf(stderr, "Blending not available\n");
        return 1;
    }
    prefsizew(WINSIZE, WINSIZE);
    winopen("blendcircs");
    mmode(MVIEWING);
    RGBmode();
    gconfig();
    mmode(MVIEWING);
    ortho2(-1.0, 1.0, -1.0, 1.0);
    glcompat(GLC_OLDPOLYGON, 0);
    blendfunction(BF_SA, BF_ONE);
    cpack(RGB_BLACK);
    clear();
    cpack(0x80000000 | RGB_RED); /* red with alpha=128/255 */
    circf(0.25, 0.0, 0.7);
    sleep(2);
    cpack(0x4f000000 | RGB_GREEN); /* green with alpha=79/255 */
    circf(-0.25, 0.25, 0.7);
    sleep(2);
    cpack(0x30000000 | RGB_BLUE); /* blue with alpha=48/255 */
    circf(-0.25, -0.25, 0.7);
    sleep(10);
    gexit();
    return 0;
}
```

```
}
```

15.3 One-Pass Antialiasing—the Smooth Primitives

Aliasing artifacts are especially objectionable in image animations, because jaggies often introduce motion unrelated to the actual direction of motion of the primitives. The techniques described in this section improve the sampling quality of primitives without requiring that the primitives be drawn more than once. These techniques therefore perform well enough to animate complex scenes.

Note: Not all systems support smoothing and not every system supports every type of smooth primitive, so refer to the man pages for details about smoothing on different systems.

Modes are provided to support the drawing of antialiased points and lines in both color map and RGB modes. Because their interactions are more critical to the antialiasing quality, antialiased polygons are supported only in the more general RGB mode. If you are drawing an image composed entirely of points and/or lines, the routines in this section are always the right choice for antialiasing. If you include polygons in the image, you should consider both the techniques described in this section and the multipass accumulation technique described in Section 15.4.

15.3.1 High-Performance Antialiased Points—`pntsmooth`

By default, IRIS-4D Series systems sample points by selecting and drawing the pixel nearest the exact projection of the mathematical point. You can enable subpixel sampling and use `pntsmooth()` to draw antialiased points.

```
subpixel(TRUE);  
pntsmooth(SMP_ON);
```

When you enable `subpixel()` mode, you defeat the default behavior of rounding projected vertices to the nearest pixel center. Exact point position is made available to the sampling hardware. By enabling `pntsmooth()` mode, you replace the default sampling of points with coverage sampling of a unit-diameter* circle centered around the exact mathematical point position. All that remains is instructing the system on how to use the computed pixel

coverage to blend between the background color and the point color at each pixel. This instruction differs based on whether the drawing is done in color map mode or in RGB mode.

When you enable `pntsmooth()` while in color map mode, the antialiasing hardware uses computed pixel coverage to replace the 4 least significant bits of the point's color. Therefore, for color map antialiased points to blend correctly, you must initialize a 16-entry colormap block (whose lowest entry location is a multiple of 16) to a ramp between the background color (lowest index) and the point color (highest index). Before drawing points, clear the background to the same color used as background in the colormap ramp.

When you draw a point with a color index in the range of the specified ramp, pixels in the area of the exact mathematical point are written with color indices that select ramp values based on the fraction of the pixel that is obscured by the point's unit-diameter circle. Because the sampling hardware modifies only the 4 least significant bits of the point's color, you can initialize and use multiple color ramps, each with a different point color, in the same image. Note that all ramps must blend to the same background color, which must be the color of the background used for the image.

This sample program, *pnt.cm.c*, illustrates the difference in image quality when you use `pntsmooth()` and `subpixel()` together to antialias color map points. The antialiased points and lines drawn by these example programs look better if you set gamma correction to 2.4, instead of the default value of 1.7.

```
/*
 * Drag a string of color map antialiased points with the cursor.
 * Disable antialiasing while the left mouse button is depressed.
 * Disable subpixel positioning while the middle mouse button is depressed.
 */

#include <stdio.h>
#include <gl/gl.h>
#include <gl/device.h>

#define WINSIZE 400
#define RAMPBASE 64 /* avoid the first 64 colors */
#define RAMPSIZE 16
```

* `pntsize()` and `pntsizef()` also affect antialiased points, but may not support the range of sizes supported by aliased points. See the *pntsize(3G)* man page for details.

```
#define RAMPSTEP (255 / (RAMPSIZE-1))  
#define MAXPOINT 25  
  
Device devs[2] = {MOUSEX,MOUSEY};
```

```

main()
{
    short val, vals[2];
    long i, xorg, yorg;
    float vert[2], x, y, interp;
    if (getgdesc(GD_PNTSMOOTH_CMODE) == 0) {
        fprintf(stderr, "Color map mode point antialiasing not available\n");
        return 1;
    }
    if (getgdesc(GD_BITS_NORM_DEL_CMODE) < 8) {
        fprintf(stderr, "Need 8 bitplanes in doublebuffer color map mode\n");
        return 1;
    }
    prefsize(WINSIZE, WINSIZE);
    winopen("pntsmooth.index");
    mmode(MVLEWING);
    ortho2(-0.5, WINSIZE-0.5, -0.5, WINSIZE-0.5);
    doublebuffer();
    gconfig();
    qdevice(ESCKEY);
    qdevice(LEFTMOUSE);
    qdevice(MIDDLEMOUSE);
    getorigin(&xorg, &yorg);
    for (i = 0; i < RAMPSTEP; i++)
        mapcolor(i + RAMPBASE, i * RAMPSTEP, i * RAMPSTEP, i * RAMPSTEP);
    while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
        color(RAMPBASE);
        clear();
        getdev(2, devs, vals);
        x = vals[0] - xorg;
        y = vals[1] - yorg;
        pntsmooth(getbutton(LEFTMOUSE) ? SMP_OFF : SMP_ON);
        subpixel(getbutton(MIDDLEMOUSE) ? FALSE : TRUE);
        color(RAMPBASE+RAMPSTEP-1);
        bgnpoint();
        for (i=0; i<=MAXPOINT; i++) {
            interp = (float)i / (float)MAXPOINT;
            vert[0] = 100.0 * interp + x * (1.0 - interp);
            vert[1] = 100.0 * interp + y * (1.0 - interp);
            v2f(vert);
        }
        endpoint();
        swapbuffers();
    }
    gexit();
    return 0;
}

```

```
}
```

Notice how smoothly the antialiased points move as you move the cursor. Now defeat the antialiasing by pressing the left mouse button, and notice that the points move less smoothly, and that they do not line up nearly as well as the antialiased points. The image quality degrades in exactly the same way when you defeat subpixel positioning by pressing the middle mouse button.

The antialiased points look good when they are not drawn touching each other. However, when you move the cursor near the lower-left corner of the window, causing the points to bunch together, the image quality again degrades. This is because pixels that are obscured by more than one point take as their value the color computed for the last point drawn. There is no general solution to the problem of overlapping primitives while drawing in color map mode.

The problem of overlapping primitives is handled well when antialiasing in RGB mode. When you enable `pntsmooth()` in RGB mode, the antialiasing hardware uses computed pixel coverage to scale the alpha value of the point's color. If the alpha value of the incoming point is 1.0, scaling it by the computed pixel coverage results in a pixel alpha value that is directly proportional to pixel coverage. For RGB antialiased points to draw correctly, set `blendfunction()` to merge new pixel color components into the frame buffer using the incoming alpha value.

This sample program, *pnt.rgb.c*, illustrates RGB mode point antialiasing.

```
/*
 * Drag a string of RGB antialiased points with the cursor.
 * Change from a merge-blend to an accumulate-blend when the left
 * mouse button is depressed. Use the "smoother" antialiasing sampling
 * algorithm when the middle mouse button is depressed.
 */

#include <stdio.h>
#include <gl/gl.h>
#include <gl/device.h>

#define WINSIZE 400
#define MAXPOINT 25

Device devs[2] = {MOUSEX,MOUSEY};

main()
{
```

```

short val, vals[2];
long i, xorg, yorg;
float vert[2], x, y, interp;

if (getgdesc(GD_PNTSMOOTH_RGB) == 0) {
    fprintf(stderr, "RGB mode point antialiasing not available\n");
    return 1;
}
prefsize(WINSIZE, WINSIZE);
winopen("pntsmooth.rgb");
mmode(MVIEWING);
ortho2(-0.5, WINSIZE-0.5, -0.5, WINSIZE-0.5);
doublebuffer();
RGBmode();
gconfig();
qdevice(ESCKEY);
qdevice(LEFTMOUSE);
qdevice(MIDDLEMOUSE);
getorigin(&xorg, &yorg);
subpixel(TRUE);
while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
    cpack(0);
    clear();
    getdev(2, devs, vals);
    x = vals[0] - xorg;
    y = vals[1] - yorg;
    cpack(0xffffffff);
    blendfunction(BF_SA, getbutton(LEFTMOUSE) ? BF_ONE : BF_MSA);
    pntsmooth(getbutton(MIDDLEMOUSE) ? (SMP_ON | SMP_SMOOTHER) : SMP_ON);
    bgnpoint();
    for (i=0; i<=MAXPOINT; i++) {
        interp = (float)i / (float)MAXPOINT;
        vert[0] = 100.0 * interp + x * (1.0 - interp);
        vert[1] = 100.0 * interp + y * (1.0 - interp);
        v2f(vert);
    }
    endpoint();
    swapbuffers();
}
gexit();
return 0;
}

```

Unlike the color map antialiased points, the RGB antialiased points look good when they are bunched together. This is because RGB blending allows multiple points to contribute to a single pixel in a meaningful way.

In this demonstration a blend function that interpolates between the incoming and frame buffer color components, based on the incoming alpha, is used by default.

```
blendfunction(BF_SA,BF_MSA)
```

Press the left mouse button to switch to a blend function that simply accumulates color, again scaled by incoming alpha:

```
blendfunction(BF_SA,BF_ONE)
```

The difference between `blendfunction(BF_SA,BF_MSA)` and `blendfunction(BF_SA,BF_ONE)` is more apparent when you draw lines (see Section 15.3.2). In this demonstration, note that bunched points are a little brighter when you select the accumulating blending function.

You can switch from the standard antialiasing sampling algorithm to a “smoother” algorithm by pressing the middle mouse button. Not all systems support the higher-quality point sampling algorithm. Refer to the `pntsmooth()` manual page for details. This algorithm modifies more pixels per antialiased point than does the standard antialiasing algorithm. As a result, it produces slightly higher-quality antialiased points, at the price of slightly reduced performance. Set the “smoother” algorithm by calling

```
pntsmooth(SMP_ON | SMP_SMOOTHER);
```

Because RGB mode antialiased points are blended into the frame buffer, they can be drawn in any color and over any background. Unless you want to draw transparent, antialiased points, however, be sure to specify alpha as 1.0 when drawing antialiased RGB points.

15.3.2 High-Performance Antialiased Lines—`linesmooth`

By default, IRIS-4D Series systems sample lines by selecting and drawing the pixels nearest the projection of the mathematical line. You can enable subpixel sampling and use `linesmooth()` to draw antialiased lines.

```
subpixel(TRUE);  
linesmooth(SML_ON);
```


By enabling `linesmooth()` mode, you replace the default sampling of lines with coverage sampling of a unit-width* rectangle centered around the exact mathematical line. All that remains is instructing the system on how to use the computed pixel coverage to blend between the background color and the line color at each pixel. This instruction differs based on whether the drawing is done in color map mode or in RGB mode.

When you enable `linesmooth()` while in color map mode, the antialiasing hardware uses computed pixel coverage to replace the 4 least significant bits of the line's color. Therefore, for color map antialiased lines to appear correct, you must initialize a 16-entry colormap block (whose lowest entry location is a multiple of 16) to a ramp between the background color (lowest index) and the line color (highest index). Before drawing lines, clear the background to the same color used as background in the color map ramp.

When you draw a line with a color index in the range of the specified ramp, pixels in the area of the exact mathematical line are written with color indices that select ramp values based on the fraction of the pixel that is obscured by the line's unit-width rectangle. Because the sampling hardware modifies only the 4 least significant bits of the line's color, you can initialize and use multiple color ramps, each with a different line color, in the same image. Note that all ramps must blend to the same background color, which must be the color of the background used for the image.

Note: To improve antialiasing performance on the IRIS Indigo, set `zsource(ZRC_COLOR)`.

* `linewidth()` and `linewidthf()` also affect antialiased lines, but may not support the range of sizes supported by aliased lines. See the `linewidth(3G)` man page for details.

This sample program, *line.cm.c*, illustrates the difference in image quality when you use `linesmooth()` and `subpixel()` together to antialias color map lines. The program draws a single straight line, made up of several individual line segments.

```
/*
 *Drag a string of color map antialiased line segments with the cursor.
 *Disable antialiasing while the left mouse button is depressed.
 *Disable subpixel positioning while the middle mouse button is depressed.
 */

#include <stdio.h>
#include <gl/gl.h>
#include <gl/device.h>

#define WINSIZE 400
#define RAMPBASE 64 /* avoid the first 64 colors */
#define RAMPSTEP 16
#define RAMPSTEP (255 / (RAMPSTEP-1))
#define MAXVERTEX 10

Device devs[2] = {MOUSEX,MOUSEY};

main()
{
    short val, vals[2];
    long i, xorg, yorg;
    float vert[2], x, y, interp;

    if (getgdesc(GD_LINESMOOTH_CMODE) == 0) {
        fprintf(stderr, "Color map mode line antialiasing not available\n");
        return 1;
    }
    if (getgdesc(GD_BITS_NORM_DBL_CMODE) < 8) {
        fprintf(stderr, "Need 8 bitplanes in doublebuffer color map mode\n");
        return 1;
    }
    prefsizew(WINSIZE, WINSIZE);
    winopen("linesmooth.index");
    mmode(MVIEWING);
    ortho2(-0.5, WINSIZE-0.5, -0.5, WINSIZE-0.5);
    doublebuffer();
    gconfig();
    qdevice(ESCKEY);
    qdevice(LEFTMOUSE);
    qdevice(MIDDLEMOUSE);
}
```

```

getorigin(&xorg, &yorg);
for (i = 0; i < RAMPSIZE; i++)
    mapcolor(i + RAMPBASE, i * RAMPSTEP, i * RAMPSTEP, i * RAMPSTEP);

while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
    color(RAMPBASE);
    clear();
    getdev(2, devs, vals);
    x = vals[0] - xorg;
    y = vals[1] - yorg;
    linesmooth(getbutton(LEFTMOUSE) ? SML_OFF : SML_ON);
    subpixel(getbutton(MIDDLEMOUSE) ? FALSE : TRUE);
    color(RAMPBASE+RAMPSIZE-1);
    bgnline();
    for (i=0; i<=MAXVERTEX; i++) {
        interp = (float)i / (float)MAXVERTEX;
        vert[0] = 100.0 * interp + x * (1.0 - interp);
        vert[1] = 100.0 * interp + y * (1.0 - interp);
        v2f(vert);
    }
    endlime();
    swapbuffers();
}
gexit();
return 0;
}

```

Notice how smooth the edges of the antialiased lines are, and how smoothly they move as you move the cursor. Now defeat the antialiasing by pressing the left mouse button, and notice that the lines become jagged. When you defeat subpixel positioning by pressing the middle mouse button, the individual line segments that make up the long line remain antialiased, but they no longer combine to form a single straight line. This is because the endpoints of the segments have been coerced to the nearest pixel centers, which are rarely on the exact mathematical line. Thus, you can antialias lines, unlike points, while `subpixel()` mode is `FALSE`. However, the image quality is still greatly enhanced when you enable subpixel positioning of vertices.

Like color map antialiased points, color map antialiased lines interact poorly when they intersect on the screen. The problem of overlapping primitives is handled well when antialiasing in RGB mode. When you enable `linesmooth()` in RGB mode, the antialiasing hardware uses computed pixel coverage to scale the alpha value of the line's color. If the alpha value of the incoming line is 1.0, scaling it by the computed pixel coverage results in a pixel alpha value that is directly proportional to pixel coverage.

For RGB antialiased lines to draw correctly, set `blendfunction()` to merge new pixel color components into the frame buffer using the incoming alpha value.

This sample program, *line.rgb.c*, illustrates RGB mode line antialiasing:

```
/*
 * Rotate a pinwheel of antialiased lines drawn in RGB mode.
 * Change to the "smoother" sampling function when the left mouse button
 * is depressed.
 * Change to the "end-corrected" sampling function when the middle mouse
 * button is depressed.
 * Change to a "color index like" blend function when the i-key is pressed.
 * Change from merge-blend to accumulate-blend when the a-key is depressed.
 * Disable subpixel positioning when the s-key is depressed.
 */

#include <stdio.h>
#include <gl/gl.h>
#include <gl/device.h>

#define WINSIZE 400
#define MAXLINE 48
#define ROTANGLE (360.0 / MAXLINE)

float vert0[2] = {0.0,0.0};
float vert1[2] = {0.8,0.0};

main()
{
    int i;
    int smoothmode;
    short val;

    if (getgdesc(GD_LINESMOOTH_RGB) == 0) {
        fprintf(stderr, "RGB mode line antialiasing not available\n");
        return 1;
    }
    prefsizew(WINSIZE, WINSIZE);
    winopen("linesmooth.rgb");
    mmode(MVIEWING);
    ortho2(-1.0,1.0,-1.0,1.0);
    doublebuffer();
    RGBmode();
    gconfig();
    qdevice(ESCKEY);
}
```

```

qdevice(LEFTMOUSE);
qdevice(MIDDLEMOUSE);

while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
    cpack(0);
    clear();
    cpack(0xffffffff);
    smoothmode = SML_ON;
    if (getbutton(LEFTMOUSE))
        smoothmode |= SML_SMOOTHER;
    if (getbutton(MIDDLEMOUSE))
        smoothmode |= SML_END_CORRECT;
    linesmooth(smoothmode);
    if (getbutton(IKEY))
        blendfunction(BF_SA,BF_ZERO);
    else if (getbutton(AKEY))
        blendfunction(BF_SA,BF_ONE);
    else
        blendfunction(BF_SA,BF_MSA);
    subpixel(getbutton(SKEY) ? FALSE : TRUE);
    pushmatrix();
    rot(getvaluator(MOUSEX) / 25.0, 'z');
    for (i=0; i<MAXLINE; i++) {
        bgnline();
        v2f(vert0);
        v2f(vert1);
        endline();
        rot(ROTANGLE, 'z');
    }
    popmatrix();
    swapbuffers();
}
gexit();
return 0;
}

```

Notice that the RGB mode antialiased lines look good where they intersect at the center of the pinwheel. This is because RGB blending allows multiple lines to contribute to a single pixel in a meaningful way. In this demonstration a blend function that interpolates between the incoming and frame buffer color components, based on the incoming alpha, is used by default:

```
blendfunction(BF_SA,BF_MSA)
```

You can switch to a blend function that simply accumulates color, again scaled by incoming alpha, by pressing the **<A>** key:

```
blendfunction(BF_SA,BF_ONE)
```

This blending function makes the slight noise at the center of the pinwheel disappear, because these pixels all accumulate and clamp at full brightness. This technique works well with white lines on a black background, but does not do well in other situations.

You can simulate the appearance of color map mode lines by pressing the **<I>** key, which forces a blending function that overwrites pixels:

```
blendfunction(BF_SA,BF_ZERO);
```

When you defeat subpixel positioning of line endpoints by pressing the **<S>** key, the pinwheel ceases to behave like a rigid object, and instead appears to wiggle and twist as it is rotated.

You can switch from the standard antialiasing sampling algorithm to a “smoother” algorithm by pressing the left mouse button. Not all systems support the higher-quality line sampling algorithm. Refer to the man page for details. This algorithm modifies more pixels per unit line length than does the standard antialiasing algorithm. As a result, it produces slightly higher-quality antialiased lines, at the price of slightly reduced performance. Set the “smoother” algorithm by calling `linesmooth (SML_ON | SML_SMOOTHER)`.

Notice that when it is selected, lines at all angles appear to have the same width, and the “cloverleaf” pattern at the center of the pinwheel disappears. When you rotate the pinwheel with the left mouse button pressed, the only image artifact that remains is the sudden changing of line length observed at the ends of the lines. Press the middle mouse button to select a sampling algorithm that correctly samples line length as well as line cross-section. Invoke this “end-corrected” algorithm by calling `linesmooth(SML_ON | SML_END_CORRECT)`.

When you select both “smoother” and “end-correct”, the rotating pinwheel appears absolutely rigid, with even width lines and no jagged edges.

Because RGB antialiased lines are blended into the frame buffer, they can be drawn in any color over any background. Unless you want to draw transparent, anti-aliased lines, however, be sure to specify alpha as 1.0 when drawing antialiased RGB lines.

Note: Because RGB antialiased lines are blended, they interact well at intersections. However, when two RGB antialiased lines are drawn between the same vertices, the line quality is reduced noticeably. When the polygons in a standard geometric model are drawn as lines, either explicitly or using `polymode`, lines at the edges of adjacent polygons are drawn twice, and therefore do not antialias well in RGB mode. For best results, modify the database traversal so that edges of adjacent polygons are drawn only once.

15.3.3 High-Performance Antialiased Polygons—`polysmooth`

By default, IRIS-4D Series systems sample polygons by selecting and drawing the pixels whose exact center points are within the boundary described by the projection of the mathematical polygon edges.

You can enable subpixel sampling and use `polysmooth()` to draw antialiased polygons.

```
subpixel(TRUE);
polysmooth(PYSM_ON);
```

When you enable subpixel mode, you defeat the default behavior of rounding projected vertices to the nearest pixel center. Exact polygon vertex positions are made available to the sampling hardware. By enabling `polysmooth()` mode, you replace the default sampling of polygons with coverage sampling—the fraction of each pixel covered by the polygon is computed. All that remains is instructing the system how to use the computed pixel coverage to blend between the background color and the polygon color at each pixel. Because this blending operation is more critical for polygon antialiasing than it is for point or line antialiasing, polygon antialiasing is supported only in RGB mode, not in color map mode.

This sample program, *poly.rgb.c*, draws a single antialiased triangle:

```
/*
 * Rotate a single antialiased triangle drawn in RGB mode.
 * Disable antialiasing when the left mouse button is depressed.
 * Disable subpixel positioning when the middle mouse button is depressed.
 */

#include <stdio.h>
#include <gl/gl.h>
#include <gl/device.h>
```

```

#define WINSIZE 400

float vert0[2] = {0.0,0.0};
float vert1[2] = {0.8,0.0};
float vert2[2] = {0.4,0.4};

main()
{
    short val;

    if (getgdesc(GD_POLYSMOOTH) == 0) {
        fprintf(stderr, "polygon antialiasing not available\n");
        return 1;
    }
    prefsizew(WINSIZE, WINSIZE);
    winopen("polysmooth.rgb");
    mmode(MVIEWING);
    ortho2(-1.0,1.0,-1.0,1.0);
    doublebuffer();
    RGBmode();
    gconfig();
    qdevice(ESCKEY);
    qdevice(LEFTMOUSE);
    qdevice(MIDDLEMOUSE);
    blendfunction(BF_SA,BF_MSA);
    while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
        cpack(0);
        clear();
        cpack(0xffffffff);
        polysmooth(getbutton(LEFTMOUSE) ? PYSM_OFF : PYSM_ON);
        subpixel(getbutton(MIDDLEMOUSE) ? FALSE : TRUE);
        pushmatrix();
        rot(getvaluator(MOUSEX) / 25.0, 'z');
        rot(getvaluator(MOUSEY) / 10.0, 'x');
        bgnpolygon();
            v2f(vert0);
            v2f(vert1);
            v2f(vert2);
        endpolygon();
        popmatrix();
        swapbuffers();
    }
    gexit();
    return 0;
}

```


Move the cursor left and right to rotate the triangle, and note the smoothness of its edges. When you move the cursor toward the top of the screen, the triangle rotates away from you until it becomes perpendicular to your viewing direction. Note that when it is perpendicular, it disappears completely. This is because the projection of a triangle on edge covers no area on the screen, and therefore all pixel coverages are zero.

When you press the left mouse button, the triangle is drawn aliased. When you press the middle mouse button, the triangle vertices are no longer subpixel-positioned. Notice that the edges remain smooth, but that the triangle motion is no longer smooth, and the triangle no longer appears rigid.

This simple example of a single antialiased triangle drawn on a black background works correctly with the standard blending function:

```
blendfunction(BF_SA,BF_MSA)
```

However, when multiple antialiased triangles are drawn with adjacent edges, the standard blending function no longer produces good results.

This sample program, *poly2.rgb.c*, draws a bowtie-shaped object, constructed of four triangles in a planar mesh:

```
/*
 * Rotate a patch of antialiased triangles drawn in RGB mode.
 * Disable special polygon-blend when the left mouse button is depressed.
 * Disable subpixel positioning when the middle mouse button is depressed.
 */

#include <stdio.h>
#include <gl/gl.h>
#include <gl/device.h>

#define WINSIZE 400

float vert0[2] = {0.0,0.0};
float vert1[2] = {0.0,0.4};
float vert2[2] = {0.4,0.1};
float vert3[2] = {0.4,0.3};
float vert4[2] = {0.8,0.0};
float vert5[2] = {0.8,0.4};
```

```

main()
{
    short val;

    if (getgdesc(GD_POLYSMOOTH) == 0) {
        fprintf(stderr, "polygon antialiasing not available\n");
        return 1;
    }
    prefsizew(WINSIZE, WINSIZE);
    winopen("polysmooth2.rgb");
    mmode(MVIEWING);
    ortho(-1.0,1.0,-1.0,1.0,-1.0,1.0);
    doublebuffer();
    RGBmode();
    gconfig();
    qdevice(ESCKEY);
    qdevice(LEFTMOUSE);
    qdevice(MIDDLEMOUSE);
    polysmooth(PYSM_ON);
    shademodel(FLAT);
    while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
        cpack(0);
        clear();
        cpack(0xffffffff);
        if (getbutton(LEFTMOUSE))
            blendfunction(BF_SA,BF_MSA);
        else
            blendfunction(BF_MIN_SA_MDA,BF_ONE);
        subpixel(getbutton(MIDDLEMOUSE) ? FALSE : TRUE);
        pushmatrix();
        rot(getvaluator(MOUSEX) / 25.0, 'z');
        rot(getvaluator(MOUSEY) / 10.0, 'x');
        bgnmesh();
        v2f(vert0);
        v2f(vert1);
        v2f(vert2);
        v2f(vert3);
        v2f(vert4);
        v2f(vert5);
        endtmesh();
        popmatrix();
        swapbuffers();
    }
    gexit();
    return 0;
}

```

Notice that the internal edges of the four triangles that make up the bowtie are visible. Press the left mouse button, enabling the special polygon blending function, and note that these internal edges disappear. They are visible when you use the standard blending function because the standard blending function operates with uncorrelated coverages, such as those generated by antialiased points and lines.

Two adjacent polygons generate pixel coverages that are highly correlated—they always sum to 100% for pixels covered by the shared edge—and are therefore inappropriate for the standard blending function. Consider, for example, a pixel that is covered 60% by the first polygon that intersects it, and 40% by a second polygon adjacent to the first. Assuming white polygons and a black background, the first polygon raises the pixel intensity to 0.6, which is the correct value. However, the second polygon raises the pixel intensity to only 0.76, rather than to 1.0 as is desired. This is because the standard blending function assumes that the 60% and 40% coverages are uncorrelated, so 60% of the additional 40% is assumed to have been covered by the original 60%. Thus in uncorrelated coverage arithmetic, 60% plus 40% equals 76%, not 100%.

The special blending function `blendfunction(BF_MIN_SA_MDA, BF_ONE)` works with correlated coverages, the kind generated by antialiased polygon images. As the example code illustrated, the correlated blend does a good job with polygonal data. It is, however, much more difficult to use correlated blending than uncorrelated blending.

The requirements for using the correlated blending function are:

- You must have alpha bitplanes.
- You must draw polygons in order from the nearest to the farthest (not farthest to nearest as in the other antialiasing examples).
- You must not draw backfacing polygons (use `backface()`).
- The background color bitplanes, including the alpha bitplanes, must be cleared to zero before drawing starts.
- If the background is any color other than black, it must be filled as a polygon (i.e. not with a `clear()` command) after all polygons are drawn.
- You must draw all primitives (points, lines, and polygons) using the correlated blending function.

The correlated blending function works by accumulating pixel coverage in the frame buffer alpha bitplanes. The coverage granted each pixel write is limited by the total remaining at that pixel. When no coverage is left, additional writes to that pixel are ignored.

Because polygons must be drawn in depth-sorted order, you cannot use the z-buffer to eliminate hidden surfaces. Thus, polygon antialiasing, unlike point and line antialiasing, requires significant changes to the way the object data are traversed. It is therefore more difficult to use than are point and line antialiasing. If performance is not an absolute requirement, the accumulation buffer technique described in Section 15.4 is a better choice for polygon antialiasing.

This sample program, *poly3.rgb.c*, demonstrates correct polygon antialiasing of two cubes against a non-black background:

```
/*
 * Rotate two antialiased cubes in RGB mode.
 * Disable antialiasing by depressing the left mouse button.
 */

#include <stdio.h>
#include <gl/gl.h>
#include <gl/device.h>

#define WINSIZE 400
#define SIZE (0.2)
#define OFFSET(0.5)
#define CUBE0 OFFSET
#define CUBE1 (-OFFSET)

float vert0[4] = {-SIZE, -SIZE, SIZE};
float vert1[4] = { SIZE, -SIZE, SIZE};
float vert2[4] = {-SIZE, SIZE, SIZE};
float vert3[4] = { SIZE, SIZE, SIZE};
float vert4[4] = {-SIZE, SIZE, -SIZE};
float vert5[4] = { SIZE, SIZE, -SIZE};
float vert6[4] = {-SIZE, -SIZE, -SIZE};
float vert7[4] = { SIZE, -SIZE, -SIZE};

float cvert0[2] = {-1.0, -1.0};
float cvert1[2] = { 1.0, -1.0};
float cvert2[2] = { 1.0, 1.0};
float cvert3[2] = {-1.0, 1.0};
```

```

main()
{
    short val;
    float xang;

    if (getgdesc(GD_POLYSMOOTH) == 0) {
        fprintf(stderr, "polygon antialiasing not available\n");
        return 1;
    }
    prefsizewin(WINSIZE, WINSIZE);
    winopen("polysmooth3.rgb");
    mmode(MVIEWING);
    ortho(-1.0,1.0,-1.0,1.0,-1.0,1.0);
    doublebuffer();
    RGBmode();
    gconfig();
    device(ESCKEY);
    qdevice(LEFTMOUSE);
    blendfunction(BF_MIN_SA_MDA,BF_ONE);
    subpixel(TRUE);
    backface(TRUE);
    shademodel(FLAT);

    while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
        cpack(0);
        clear();
        polysmooth(getbutton(LEFTMOUSE) ? PYSM_OFF : PYSM_ON);
        pushmatrix();
        xang = getvaluator(MOUSEY) / 5.0;
        rot(xang, 'x');
        rot(getvaluator(MOUSEX) / 5.0, 'z');
        if (xang < 90.0) {
            drawcube(CUBE0);
            drawcube(CUBE1);
        } else {
            drawcube(CUBE1);
            drawcube(CUBE0);
        }
        popmatrix();
        drawbackground();
        swapbuffers();
    }
    gexit();
    return 0;
}

```

```

drawcube(offset)
float offset;
{
    pushmatrix();
    translate(0.0,0.0,offset);
    bgntmesh();
        v3f(vert0);
        v3f(vert1);
        cpack(0xff0000ff);
        v3f(vert2);
        v3f(vert3);
        cpack(0xff00ff00);
        v3f(vert4);
        v3f(vert5);
        cpack(0xffff0000);
        v3f(vert6);
        v3f(vert7);
        cpack(0xff00ffff);
        v3f(vert0);
        v3f(vert1);
    endtmesh();
    bgntmesh();
        v3f(vert0);
        v3f(vert2);
        cpack(0xffff00ff);
        v3f(vert6);
        v3f(vert4);
    endtmesh();
    bgntmesh();
        v3f(vert1);
        v3f(vert7);
        cpack(0xffffffff00);
        v3f(vert3);
        v3f(vert5);
    endtmesh();
    popmatrix();
}

drawbackground() {
    cpack(0xffffffff);
    bgnpolygon();
        v2f(cvert0);
        v2f(cvert1);
        v2f(cvert2);
        v2f(cvert3);
    endpolygon();
}

```

Note that the nearer cube is drawn first, that cube faces are not sorted because back face elimination handles the sorting of convex solids, and that the background is drawn last as a single polygon.

When you press the left mouse button, antialiasing is disabled, but the correlated blend function remains enabled. Otherwise, the drawing order of the primitives would have to be changed.

15.4 Multipass Antialiasing with the Accumulation Buffer

This section describes techniques for computing pixel area coverage for various primitives, and using this coverage information to blend pixels into the framebuffer. This technique, called *accumulation*, is an iterative process that converges on a very accurate image. It easily handles all combinations of points, lines, and polygons, but it cannot typically be used to generate an interactive image. Accumulation also has applications in other advanced rendering techniques.

Accumulation is somewhat like blending, in that multiple images are composited to produce the final image. It differs from blending, however, in that its operation is completely separated from the rendering of a single frame. The accumulation buffer is an extended range bitplane bank in the normal frame buffer. You do not draw images into it; rather, images drawn in the front or back buffer of the normal frame buffer are added to the contents of the accumulation buffer after they are completely rendered.

15.4.1 Configuring the Accumulation Buffer

Before you can use the accumulation buffer, you must allocate bitplanes for it. `acsize()` specifies the number of bitplanes to be allocated for each color component in the accumulation buffer:

```
void acsize(long planes)
```

The number of bits that can be allocated to the accumulation buffer varies, depending on the system type. Sizes of 0 and 16 are used for most IRIS-4D systems, IRIS Indigo uses 32, and RealityEngine uses either a 12-bit unsigned or 24-bit signed accumulation buffer.

Color components in the accumulation buffer are signed values, so the range for each component depends on the size of the accumulation buffer. For example, a 16-bit accumulation buffer actually allocates 64 bitplanes, 16 each for red, green, blue, and alpha. You must call `gconfig()` after `acsize()` to activate the new specification.

15.4.2 Using the Accumulation Buffer

After bitplanes have been allocated for the accumulation buffer, you can use the `acbuf()` command to add the contents of the front or back bitplanes of the normal frame buffer to the accumulation buffer, and to return the accumulation buffer contents to either the front or back bitplanes. Call `acbuf()` only while the normal frame buffer is in RGB mode.

`acbuf()` operates on the accumulation buffer, which must already have been allocated using `acsize()` and `gconfig()`. When *op* is `AC_CLEAR`, each component of the accumulation buffer is set to *value*. When *op* is `AC_ACCUMULATE`, pixels are taken from the current `readsource()` (front, back, or z-buffer). Pixel components red, green, blue and alpha are each scaled by *value*, which must be in the range $-256.0 \leq \textit{value} \leq 256.0$, and added to the current contents of the accumulations buffer.

Finally, when *op* is `AC_RETURN`, pixels are taken from the accumulation buffer. Each pixel component is scaled by *value*, which must be in the range 0.0 through 1.0, clamped to the integer range 0 through 255, and returned to the currently active drawing buffer (as specified by the `frontbuffer()`, `backbuffer()`, and `zdraw` commands). All special pixel operations—including z-buffer, blending function, logical operation, stenciling, and texture mapping—are ignored during this transfer. (These commands implement several other accumulation buffer operations. See the man pages for details on these operations.)

Accumulation buffer pixels map one-to-one with pixels in the window. All accumulation buffer operations affect the pixels within the viewport, limited by the screen mask and by the edges of the window itself. Like front, back, and z-buffer pixels, accumulation buffer pixels corresponding to window pixels that are obscured by another window, or are not on the screen, are undefined.

You can use the accumulation buffer to average many renderings of the same scene into one final image. By jittering the viewing frustum slightly for each

image, you can produce a single antialiased image as the result of many averaged images. For this to work, you must:

1. Completely render the image for each pass, including using the z-buffer to eliminate hidden surfaces, if appropriate.
2. Enable subpixel positioning of all primitives used (see `subpixel`).
3. Slightly perturb the viewing frustum before rendering each image. By slightly perturbing the projection transformation before rendering each image, you can effectively move the sample position in each pixel away from the pixel center. This is particularly easy to implement when you use an orthographic projection.

This sample program, *acbuf.rgb.c*, draws an antialiased circle using a 2-D orthographic projection:

```
/*
 *Draw an antialiased circle using the accumulation buffer.
 *Disable antialiasing when the left mouse button is depressed.
 *Disable subpixel positioning when the middle mouse button is depressed.
 */

#include<stdio.h>
#include<gl/gl.h>
#include<gl/device.h>

#define WINSIZE 100
#define SAMPLES 3
#define DELTA (2.0/(WINSIZE*SAMPLES))

main()
{
    long x,y;
    short val;

    if(getgdesc(GD_BITS_ACBUF)==0){
        fprintf(stderr,"accumulation buffer not available\n");
        return 1;
    }

    prefsizew(WINSIZE,WINSIZE);
    winopen("acbuf.rgb");
    mmode(MVIEWING);
    glcompat(GLC_OLDPOLYGON,0);           /*point sample the circle*/
    doublebuffer();
    RGBmode();
    acsize(16);
}
```

```

gconfig();
qdevice(ESCKEY);
qdevice(LEFTMOUSE);
qdevice(MIDDLEMOUSE);

while(!(qtest() && qread(&val) == ESCKEY && val==0)){
    subpixel(getbutton(MIDDLEMOUSE) ? FALSE : TRUE);
    if(getbutton(LEFTMOUSE)){
        drawcirc(0.0,0.0);
    }else{
        acbuf(AC_CLEAR,0.0);
        for(x=0;x<SAMPLES;x++){
            for(y=0;y<SAMPLES;y++){
                drawcirc((x-(SAMPLES/2))*DELTA,(y-(SAMPLES/2))*DELTA);
                acbuf(AC_ACCUMULATE,1.0);
            }
        }
        acbuf(AC_RETURN,1.0/(SAMPLES*SAMPLES));
    }
    swapbuffers();
}
gexit();
return 0;
}

drawcirc(xdelta,ydelta)
floatxdelta,ydelta;
{
    ortho2(-1.0+xdelta,1.0+xdelta,-1.0+ydelta,1.0+ydelta);
    cpack(0);
    clear();
    cpack(0xffffffff);
    circf(0.0,0.0,0.8);
}

```

The circle is drawn nine times on a regular three-by-three grid. After the ninth accumulation, the resulting image is returned to the back buffer, and the buffers are swapped, making the antialiased circle visible. Note that the edges of the circle are smooth, and that when you press the left mouse button, the edges become jagged (aliased). Also note the reduction in image quality when you defeat subpixel positioning by pressing the middle mouse button.

Some other points about this sample program:

- The orthographic projection is perturbed by multiples of `DELTA`, a constant that is a function of the ratio of units in orthographic coordinates to window coordinates, and of the resolution of the subsampling.

Note that you cannot use the viewport to jitter the sample points, both because the viewport is specified with integer coordinates, and because pixels near the viewport boundary would sample incorrectly.

- Old polygon mode is defeated. Otherwise the circle would be drawn with the old fill style, rather than the new point-sampled style. Point sampling and subpixel positioning are both required to use the accumulation buffer accurately.
- Each drawing pass clears the back buffer to black, then draws the circle. In general, all drawing operations (such as clearing and using the z-buffer) must be duplicated for each pass.

You can perform accumulation buffer antialiasing with perspective projections as well as orthographic projections. The following subroutines do all the arithmetic required to implement pixel jitter using the perspective and window projection calls:

```
#include <math.h>

void subpixwindow(left,right,bottom,top,near,far,pixdx,pixdy)
float left,right,bottom,top,near,far,pixdx,pixdy;

{
    short vleft,vright,vbottom,vtop;
    float xysize,yysize,dx,dy;
    int xpixels,ypixels;
    getviewport(&vleft,&vright,&vbottom,&vtop);
    xpixels = vright - vleft + 1;
    ypixels = vtop - vbottom + 1;
    xysize = right - left;
    yysize = top - bottom;
    dx = -pixdx * xysize / xpixels;
    dy = -pixdy * yysize / ypixels;
    window(left+dx,right+dx,bottom+dy,top+dy,near,far);
}

void subpixperspective(fovy,aspect,near,far,pixdx,pixdy)
Angle fovy;
```

```

float aspect, near, far, pixdx, pixdy;

{
    float fov2, left, right, bottom, top;
    fov2 = ((fovy*M_PI) / 1800) / 2.0;
    top = near / (fcos(fov2) / fsin(fov2));
    bottom = -top;
    right = top * aspect;
    left = -right;
    subpixwindow(left, right, bottom, top, near, far, pixdx, pixdy);
}

```

In many applications, you can condition use of the accumulation buffer on user input. For example, when mouse position determines view angle, you can accumulate and display a progressively higher-quality antialiased image while the mouse is stationary. At any time during an antialiasing accumulation, the contents of the accumulation buffer represent a better image than the aliased image. You might choose a sample pattern that optimizes the intermediate results, then display each intermediate result, rather than waiting for the accumulation to be complete.

The antialiasing example implements a box filter—samples are evenly distributed inside a square pixel, and each sample has the same effect on the resulting image. Antialiasing filter quality improves when the samples are distributed in a circular pattern that is larger than a pixel, perhaps with a diameter of 0.75 pixels or so. You can further improve the filter quality by shaping it as a symmetric Gaussian function, either by changing the density of sample locations within the circle, or by keeping the sample density constant and assigning different weights to the samples. The weight of a sample is specified by *value* when you call `acbuf(AC_ACCUMULATE, value)`. A circularly symmetric Gaussian filter function yields smoother edges than does a unit-size box filter.

See `/usr/people/4Dgifts/examples/acbuf` for examples of different filters.

Regardless of the filter function, fewer samples are required to achieve a given antialiasing quality level when the samples are distributed in a random fashion, rather than in regular rows and columns.

The accumulation buffer has many rendering applications other than antialiasing. For example, to limit depth of field, you can average images projected from slightly different viewpoints and directions. To produce motion blur, you can average images with moving objects rendered in different

locations along their trajectories. To implement a filter kernel, you can convolve images with `rectcopy()` and the accumulation buffer. Because the accumulation buffer operates on signed color components, and clamps these components to the display range of 0 through 255 when they are returned to the display buffer, you can implement filters with negative components in their kernels.

Additional details of the theory and use of the accumulation buffer, as well as example images, are available in “The Accumulation Buffer: Hardware Support for High-Quality Rendering,” in *SIGGRAPH'90 Conference Proceedings*, Volume 24, Number 3, August 1990.

15.5 Antialiasing on RealityEngine Systems

This section discusses advanced features that are available only on systems with RealityEngine graphics, so you may want to skip to Chapter 16 if you do not have one of these systems.

The techniques discussed in the previous sections of this chapter suggest some different ways to reduce aliasing. RealityEngine graphics offers high performance on all the traditional antialiasing methods presented in the previous sections of this chapter. In addition, RealityEngine supports real-time antialiasing through the advanced feature of *multisampling*.

The scan conversion hardware in most graphics systems samples points, lines, and polygons with a single sample located at the center of each pixel. See Chapter 2 for more information on point-sampling.

When these single-sampled primitives are rendered, aliasing artifacts can appear. Aliasing occurs because the pixels that are only partially covered by the primitive do not get colored if the center of the pixel is not covered.

Not having enough sample points within the pixel to adequately determine the amount of pixel coverage is called *undersampling*, which results in an aliased image. A sufficient sampling method accounts for the areas of all the polygons that contribute to the shading of each pixel, rather than just a single sample point. That way, the pixel can be accurately shaded to a value that represents all polygons that are visible within that pixel.

15.5.1 Multisample Antialiasing

In single-pass multisample antialiasing, up to 16 subsamples can be evaluated at each pixel without repeatedly rendering the frame and accumulating the results. Multisampling provides for greater accuracy when rendering primitives while still maintaining a high level of geometry performance. Depth and stencil values are also evaluated and stored at each subsample, if those features are enabled.

Figure 15-2 shows example dot patterns for multisampling. Each dot represents a subsample within a single pixel. The example dot patterns shown here are *not* very efficient sampling locations. The actual sample patterns used by the hardware are efficient.

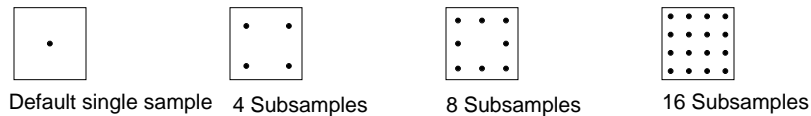


Figure 15-2 Example Multisample Patterns

15.5.2 Configuring Your System for Multisampling

In its default configuration, the standard framebuffer is configured to store a single value at each pixel. In multisampling, the multisample framebuffer is configured to store 0, 4, 8, or 16 subsamples for each pixel. The default multisample size is zero, corresponding to point-sampling.

Storage for multiple subsamples is allocated from graphics memory, as is storage for framebuffers and other features. Rather than limit the way you can use multisampling with other features by establishing fixed boundaries for memory partitions, RealityEngine allows a flexible configuration that lets you choose the combination of features that best suits your application needs.

Your RealityEngine system contains either 1, 2, or 4 Raster Manager (RM) boards. The base configuration contains one RM board. Each additional RM increases the pixel throughput, the memory for graphics features, and the amount of screen resolution available from a variety of user-selectable video formats. One additional RM lets you use either advanced graphics features such as multisampling, or lets you select a high-resolution video format. The

maximum configuration of four RMs provides both advanced graphics features and high screen resolution.

To set up multisampling, you must configure the multisample buffer with the number of samples to use. The combination of the number of RMs installed in your system, the depth of the color framebuffer (either 8 bits or 12 bits per component), the screen resolution, and the use of other features such as z-buffering, stereo buffering, or stenciling determines the number of samples available.

The framebuffer resolution that you select can affect the features that are available, such as the number of subsamples you can use, or whether color computations are performed at 8 or 12 bits per component. You can balance the trade-off between screen real estate and sample size to suit your needs.

Multisampling can be used only in RGB mode when the draw mode is `NORMALDRAW`. Because it is not possible to allocate a multisample buffer in color index mode, `multisample()` is always ignored in color index mode. When a multisample buffer is configured, multisampling is enabled by default.

Selecting the Number of Samples

Evaluating a greater number of samples provides more accuracy. You can select 0, 4, 8, or 16 samples per pixel. Use `mssize()` to configure the number of subsamples in the multisample buffer. `mssize()` takes three arguments:

<i>samples</i>	Number of subsamples to use, dependent on system configuration. Use either 0 (default single sampling), 4, 8, or 16.
<i>zsise</i>	Number of bits of z-buffer data to store at each subsample. Use either 0 or 32.
<i>ssize</i>	Number of bits of stencil data to store at each subsample. Use either 0, 1, or 8.

The GL allocates framebuffer memory when you request rendering modes. When you issue a `gconfig()`, the system attempts to honor all of your requests. If the system is unable to honor all of the requests, it may reduce the sample size or disable the use of other options such as the accumulation buffer or stereo buffering. For example, you may be granted the requested number of multisamples, but not a hardware accumulation buffer. In that case, a software accumulation buffer is substituted for the hardware accumulation buffer.

Another example is a request for 4 multisamples and stereo double buffering. If not enough resources are available to supply all of these requests, the multisample size request may be approved, but stereo buffering may be denied.

Figure 15-3 shows a conceptual diagram of how graphics memory, screen resolution, and framebuffer requests determine the configuration granted.

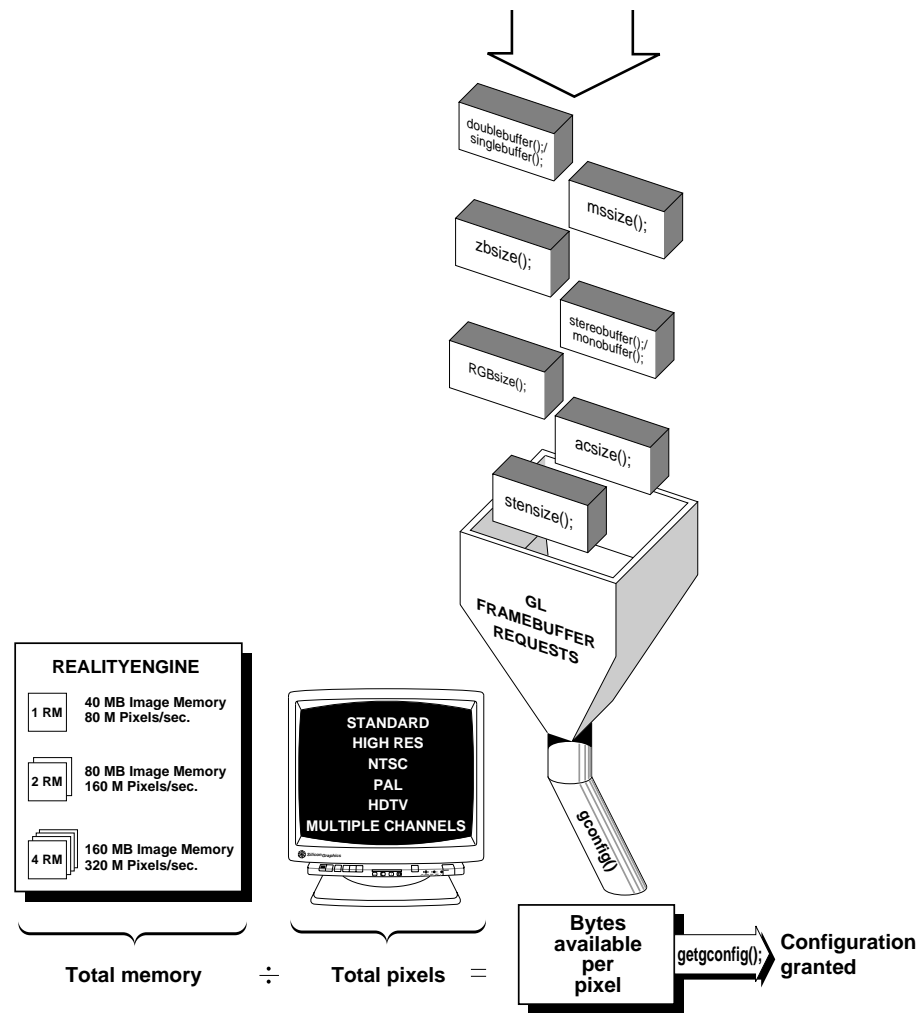


Figure 15-3 Flexible Framebuffer Configuration

The GL makes reasonable assumptions about the priority of the requests when it allocates memory.

You can query the system for the resources actually received. Use `getgconfig()` to read back the framebuffer configuration. `getgconfig()` returns the configured size of a buffer in the current drawmode.

Programs should be coded to request their most desirable configuration first, then back off and try configurations that are less stringent, but still acceptable.

The following sample code fragment illustrates this heuristic approach to configuring the RealityEngine framebuffer.

```
#include <gl/gl.h>

main()
{
    winopen("Request");

    /*this code requests 8 multisamples, but is willing to accept 4 or none*/
    RGBmode();          /* RGB mode */
    doublebuffer();     /* double buffering */
    zbsize(0);          /* deallocate main zbuffer */
    mssize(8,32,0);     /* 8 multisamples with zbuffer */
    gconfig();

    /* check for at least 4 multisamples */
    if (getgconfig(GC_MS_SAMPLES) < 4)
    {
        /* did not get enough multisamples */
        mssize(0,0,0); /* remove multisample request */
        zbsize(32);   /* restore main zbuffer */
        gconfig();
    }
}
```

Note: As illustrated in the sample code, be sure to restore the z-buffer if multisampling is not used.

Freeing Standard Framebuffer Memory

If an `mssize()` request is honored, the configuration requests for the z-buffer and stenciling apply to the multisample buffer rather than the standard framebuffer. However, memory is *not* automatically deallocated from the

standard framebuffer memory for z-buffer and stencil usage. Therefore, you should free this memory by setting the standard z-buffer size, as well as the standard stencil size, to zero. The default stencil size is already zero unless you have changed it, but the default z-buffer size is 32 bits.

Note: Unless you need the standard z-buffer in addition to the multisample buffer, deallocate memory from the standard z-buffer and stencil planes when using multisampling. The default 32-bit z-buffer on the main framebuffer uses up valuable memory that may be needed for multisampling. If multisampling is not going to be used—perhaps because a multisample request was requested, but not granted—remember to restore the z-buffer size to 32 bits.

Use `zbsize()` and `stensize()` to allocate/deallocate z-buffer and stencil memory from the standard framebuffer.

`zbsize(n)` specifies the number of z-buffer bits allocated for the standard framebuffer. When using multisampling, no z-buffer data is stored with the standard framebuffer sample.

`stensize(n)` specifies the number of stencil bits allocated for the standard framebuffer. When using multisampling, no stencil data is stored with the standard framebuffer sample.

You can use `RGBsize()` to reduce the color resolution in order to gain more memory for framebuffer requests. Specify the number of bits to be allocated per component. RealityEngine supports either 8 or 12 bits per component.

Summary

The following list summarizes the steps involved in using multisampling:

1. Set the drawing and color modes for normal drawing and RGB color:

```
drawmode(NORMALDRAW); (default)
```

```
RGBmode();
```

2. Specify the other framebuffer modes:

```
singlebuffer() (default) or doublebuffer();
```

```
monobuffer() (default) or stereobuffer();
```

3. Specify the number of samples:

```
mssize(samples, zsize, ssize);
```

4. Deallocate z-buffer and stencil bits from the main framebuffer:

```
stensize(0);
```

```
zbsize(0)
```

5. Configure the GL:

```
gconfig();
```

6. Enable multisampling:

```
multisample(TRUE); (default)
```

Note: `multisample()` can be called at any time during the rendering of a scene, except between `bgn/end` calls. Mixing multisampling and default sampling in the same scene is permitted but not recommended. The reason for this is that while the color information in the standard framebuffer is continuously updated with multisample data, the standard z-buffer is not. Therefore, z data updated at subsample locations is lost while multisampling is turned off.

15.5.3 How Multisampling Affects Rendering

The multisample buffer is a part of the color framebuffer. `multisample()` should be called only while `drawmode` is `NORMALDRAW`. Multisampling affects the results of rendering when multisampling is enabled (`TRUE`) and when the draw mode is `NORMALDRAW`.

When `multisample` is true, rendered primitives directly affect the samples in the multisample buffer. Immediately after the multisample locations at a pixel are modified, the front and/or back framebuffer colors are written with the “average” value of the multisample color values. No change is made to the standard z-buffer or stencil buffer that may be associated with the color buffers, and their contents do not affect rendering operations.

The framebuffer modes of alpha test, blending, dithering, and writemask affect the modification of the individual subsamples, and have no effect on the transfer of the average color value to the front or back color buffers. Conversely, `buffer` enables `frontbuffer()`, `backbuffer()`, `leftbuffer()`,

and `rightbuffer()` have no effect on the modification of the individual multisample locations; they affect only the transfer of the average color.

There are no special clear commands for the multisample buffer. Rather, the standard `clear()`, `zclear()`, `sclear()`, and `czclear()` commands affect the multisample buffer much as they affect the standard color, stencil, and z buffers. Clear modifies the enabled color buffers, and always modifies the color portion of each multisample location. `zclear()` operates on all multisample z locations. `sclear()` operates on all multisample stencil locations. `czclear()` behaves like `clear()/zclear()`, except that the z value is specified.

When `multisample()` is false, `polysmooth()`, `linesmooth()`, and `pntsmooth()` can be used with no performance degradation. However, unlike smooth primitives, multisampling does antialias geometry at intersection and interpenetration points.

Note: Antialiased primitives using the smooth rendering modes `pntsmooth()`, `linesmooth()`, and `polysmooth()` are superseded by multisampling— that is, these modes are undefined when multisampling is on. However, `blendfunction()` is still operational, which can adversely affect rendering performance in multisample mode. Therefore, for the best performance when multisampling, you should turn `blendfunction()` off when not performing blending. Don't use `blendfunction()` for antialiasing along with multisampling. This is especially applicable when porting previously developed code that used smooth primitives and blending for antialiasing to RealityEngine.

Multisampled Points

Until circles are implemented, points are sampled into the multisample buffer as squares centered on the exact point location.

Multisampled Lines

Lines are sampled into the multisample buffer as rectangles centered on the exact zero-area segment. The rectangle width is equal to the current linewidth. Its length is exactly equal to the length of the segment. The rectangles of colinear, abutting line segments abut exactly, so no subsamples are missed or drawn twice near the shared vertex.

Multisampled Polygons

Polygons are sampled into the multisample buffer much as they are into the standard single-sample buffer. A single color value is computed for the entire pixel, regardless of the number of subsamples at that pixel. Each multisample location is then written with this color if and only if it is geometrically within the exact polygon boundary.

If the z-buffer is enabled, the correct depth value at each multisample location is computed and used to determine whether that sample should be written or not. If stencil is enabled, the test is performed at each multisample location.

Polygon pattern bits apply equally to all multisample locations at a pixel. All sample locations are considered for modification if the pattern bit is 1. None are considered if the pattern bit is zero.

Pixels are sampled into the multisample buffer by treating each pixel as an $x_{zoom} \times y_{zoom}$ square, which is then sampled just like a polygon.

15.5.4 Using Advanced Multisample Options

You can generate special effects with advanced multisample options, which generally apply to a very specific application. Example applications include:

- Using the accumulation buffer with multisampled images.
- Using a multisample mask to select writable sample locations.
- Using alpha values to feather-blend texture edges.

Accumulating Multisampled Images

The accumulation buffer averages several samples obtained by storing multiple renderings of the same primitive, each with the primitive offset by a specific amount, a technique known as *jittering*.

Just as the default single-sample location is the center of each pixel, there are default locations for the multiple sample points, located in a cluster surrounding the center of each pixel. The default locations are chosen to produce optimum rendering quality for single-pass rendering.

Superlative image quality can be achieved when several multisampled images are composited using the accumulation buffer. Each rendering pass should use a slightly different sample pattern. These patterns have been selected to produce optimum rendering quality for the corresponding number of passes.

Accumulating multisample results can also extend the capabilities of your system. For example, if you have only enough resources to allow 4 subsamples, but you are willing to render the image twice, you can achieve the same effect as multisampling with 8 subsamples.

Use `msspattern()` to select the sample pattern. Select the sample pattern for `msspattern()` according to the number of rendering passes to accumulate. Table 15-2 lists the tokens for selecting accumulation multisample patterns.

Pattern Token	Purpose
MSP_DEFAULT	Default multisample pattern
MSP_2PASS_0	First pass of a 2-pass accumulation
MSP_2PASS_1	Second pass of a 2-pass accumulation
MSP_4PASS_0	First pass of a 4-pass accumulation
MSP_4PASS_1	Second pass of a 4-pass accumulation
MSP_4PASS_2	Third pass of a 4-pass accumulation
MSP_4PASS_3	Fourth pass of a 4-pass accumulation

Table 15-2 Tokens for Selecting Accumulation Multisample Patterns

The pattern should be changed only between complete rendering passes. It should not be changed between the time `clear()/czclear()` is called and the time that the rendered image is complete.

The following example configures the framebuffer with both a multisample buffer and an accumulation buffer for a 2-pass rendering:

```

RGBmode();
doublebuffer();
acsize(12);
mssize(4,32,0);
zbsize(0);
gconfig();
lsetdepth(getgdesc(GD_ZMAX),getgdesc(GD_ZMIN));
zfunction(ZF_GEQUAL);

```

```

zbuffer(TRUE);
multisample(TRUE);
mspattern(MSP_2PASS_0);
czclear(0,0);
/* draw the scene */
acbuf(AC_CLEAR_ACCUMULATE,1.0);
mspattern(MSP_2PASS_1);
czclear(0,0);
/* draw the scene again */
acbuf(AC_ACCUMULATE,1.0);
acbuf(AC_RETURN,0.5);
swapbuffers();

```

To maintain greater precision in the accumulation buffer, substitute the following values in the `acbuf()` commands:

```

acbuf(AC_CLEAR_ACCUMULATE,2.0);
acbuf(AC_ACCUMULATE,2.0);
acbuf(AC_RETURN,0.25);

```

or, for even greater precision, use the following values:

```

acbuf(AC_CLEAR_ACCUMULATE,4.0);
acbuf(AC_ACCUMULATE,4.0);
acbuf(AC_RETURN,0.0625);

```

This is because only 8 bits out of 12 are accumulated when using a value of 1.0, 9 bits are accumulated when using a value of 2.0, and so on—up to a maximum of 12 bits with `acsize(n)`.

Using a Multisample Mask

You can use a mask to specify a subset of multisample locations to be written at a pixel. This feature is useful for implementing fade-level-of-detail in visual simulation applications. Multisample masks can be used to perform the blending from one model to the next by rendering the additional data in the detail model using a steadily increasing percentage of subsamples as the viewpoint nears the object.

The mask specifies the ratio of writable and non-writable locations at each pixel. However, it does not single out specific locations for writing. Mask values range from 0 to 1, where 0 indicates that no locations are to be written and 1 indicates that all sample locations are to be written.

You can also set a Boolean to create the inverse mask. For example, `msmask(0.75, FALSE)` will generate a mask that allows 75% of the samples to be written, and `msmask(0.75, TRUE)` will generate a mask that allows the other 25% of the samples to be written.

Using Alpha and Color Blending Options

Multisampling can be used to solve the problem of blurred edges on textures with irregular edges, such as trees, that require extreme magnification. When the texture is magnified, the edges of the tree look artificial, as if the tree is a paper cutout. You can feather the edges to make them look more natural by including alpha in the multisample mask.

By using `msalpha()` and `afunction()` together, you can represent objects such as trees, bridges, and fences with pictures of those objects superimposed on top of a simple rectangle polygon. The see-through effect is achieved by enabling alpha blending and `afunction()` to ignore the area of the polygon not covered by the texture. See Chapter 18 to learn more about using textures.

You can use `msalpha()` to specify whether you want alpha values to be included in the multisample mask.

When `msalpha` is set to `MSA_MASK` or `MSA_MASK_ONE` while multisampling is enabled, alpha values generated by rasterization are converted to multisample masks immediately prior to the per-pixel alpha test. At each pixel, the resulting multisample mask is logically ANDed with the mask specified by `msmask`. Only multisample locations enabled by the resulting mask are considered for modification.

If `msalpha` is set to `MSA_MASK_ONE`, the alpha value presented to alpha test and to each multisample location is the maximum value supported by the framebuffer configuration, effectively 1.0.

When `msalpha` is set to `MSA_ALPHA`, no change is made to the alpha values generated by rasterization or to the mask specified by `msmask()`.

None of the multisample options—`msmask()`, `msalpha()` or `mipattern()`—has any effect when `multisample` is `FALSE`, but they are maintained for potential future use.

Chapter 16

Graphical Objects

This chapter describes the subroutines that you use to build hierarchies of drawing modules so you can draw geometry that has multiple instances of the same figure. You often want to group together a sequence of drawing subroutines and give it an identifier. The entire sequence can then be repeated with a single reference to the identifier rather than by repeating all the drawing subroutines. In the Graphics Library, such sequences are called *graphical objects*; in other systems they are sometimes known as *display lists*.

- Section 16.1, “Creating an Object,” tells you how to define the drawing modules that create an object.
- Section 16.2, “Working with Objects,” describes the subroutines you use to edit objects and mark them for special operations.

16.1 Creating an Object

A graphical object is a list of graphics primitives (drawing subroutines) to display. For example, a drawing of an automobile can be viewed as a compilation of smaller drawings of each of its parts: windows, doors, wheels, and so on. Each part (for example, a wheel) might be a graphical object—a series of `point()`, `line()`, and `polygon()` subroutines.

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

To create a graphical object, you call `makeobj()`, call the same drawing subroutines you would normally call to draw the object, and then call `closeobj()`. Between the `makeobj()` and `closeobj()` calls, drawing subroutines do not result in immediate drawing on the screen; rather, they are compiled into the object that is being created.

Thus, a graphical object is a list of primitive drawing subroutines to be executed. Drawing the graphical object consists of executing each routine in the listed order. There is no flow control, such as looping, iteration, or condition tests, except for tests that determine whether or not objects are in the viewport, as illustrated in Figure 16-3, in Section 16.2, "Working with Objects."

Note: Not all GL subroutines can be included within a graphical object. A general rule is to include drawing subroutines and not to include subroutines that return values. If you have a question about a particular routine, see the man page for that command.

makeobj

`makeobj()` creates a graphical object:

```
void makeobj (Object obj)
```

The argument *obj* is a 31-bit integer that is associated with the object. If *obj* is the number of an existing object, the contents of that object are deleted.

When `makeobj()` executes, the object number is entered into a symbol table and an empty graphical object is created. Subsequent graphics subroutines are compiled into the graphical object instead of being executed immediately. `makeobj()` creates a new object containing Graphics Library subroutines between `makeobj()` and `closeobj()`

closeobj

`closeobj()` terminates the object definition and closes the open object:

```
void closeobj(void)
```

All the subroutines in the graphical object between `makeobj()` and `closeobj()` are part of the object definition.

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

Figure 16-1 shows the sphere defined as a graphical object that is created by the following code:

```
Object obj;
makeobj(sphere=genobj());
for (phi=0; phi<PI; phi+=PI/16) {
    bgnclosedline();
    for(theta=0; theta<2*PI; theta+=PI/18) {
        vert[0] = sin(theta) * cos(phi);
        vert[1] = sin(theta) * sin(phi);
        vert[2] = cos(theta);
        v3f(vert);
    }
    endclosedline();
}
closeobj();
```

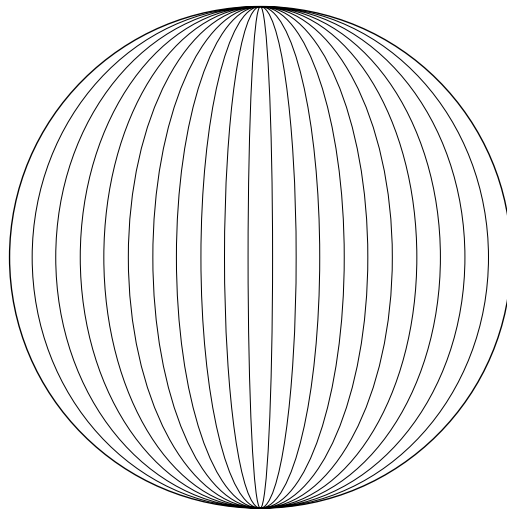


Figure 16-1 Sphere Defined as an Object

isobj

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

genobj

`genobj()` generates a unique numeric identifier:

```
Object genobj(void)
```

`genobj()` is useful in naming objects when it is impossible to anticipate what the current numeric identifier will be when the routine is called.

delobj

`delobj()` deletes an object:

```
delobj(Object obj)
```

The system frees all memory storage associated with the deleted object. The numeric identifier is undefined until it is reused to create a new object. The system ignores calls to deleted or undefined objects.

16.2 Working with Objects

You can draw, modify, and delete objects. The following sections describe those operations.

16.2.1 Drawing an Object

Once you create an object, you can draw it with a single `callobj()` command.

`callobj()` draws a created object on the screen:

```
void callobj(Object obj)
```

The argument *obj* takes the numeric identifier of the object you want to draw.

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

```
Object pearl = 1, pearls = 2
makeobj(pearl);
    color(BLUE);
    for(angle=0; angle<3600; angle=angle+300) {
        rotate(300, 'y');
        circ(0.0, 0.0, 1.0);
    }
closeobj();
makeobj(pearls);
    for(i=0; i<7; i=i+1) {
        translate(2.0, 0.0, 0.0);
        color(i);
        callobj(pearl);
    }
closeobj();
```

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

As another example of using simple objects to build more complex objects, a solar system can be defined as a hierarchical object. Calling the object *solarsystem* draws all the other objects named in its definition (the sun, the planets and their orbits).

When you call a complex object, the system draws the whole hierarchy of objects in its definition. Because the system draws the whole object *solarsystem* it can draw objects that are not visible in the viewport.

Operations known as *pruning* and *culling* guarantee that only the objects that fit within the viewport are drawn. Culling determines which parts of the picture are less than the minimum feature size, and thus too small to draw on the screen. Pruning calculates whether an object is completely outside the viewport.

Figure 16-2 shows the solar system. The diagram below the solar system is a hierarchy diagram, also called a *tree*. Branches in the tree represent calling subroutines.

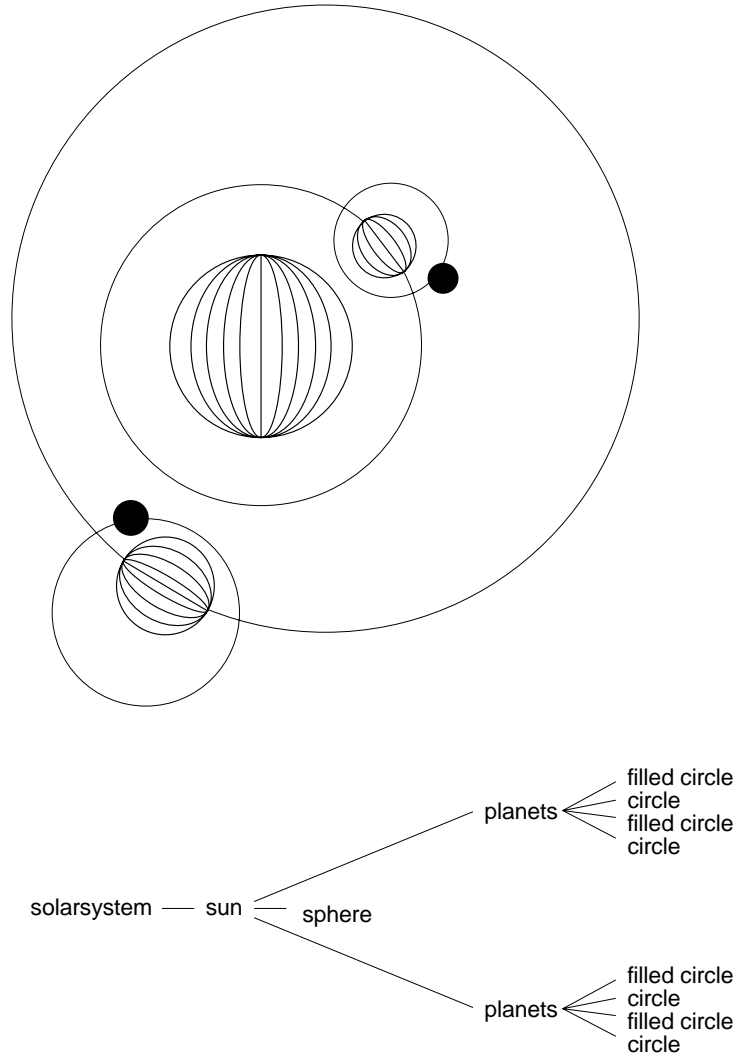


Figure 16-2 Drawing a Hierarchical Object

16.2.2 Bounding Boxes

Bounding boxes can be used to surround objects with irregular surfaces to make it easier to test them for pruning and culling.

Figure 16-3 shows some of the *solarsystem* objects surrounded by their bounding boxes. The bounding boxes can perform pruning to determine which objects are partially within the viewport.

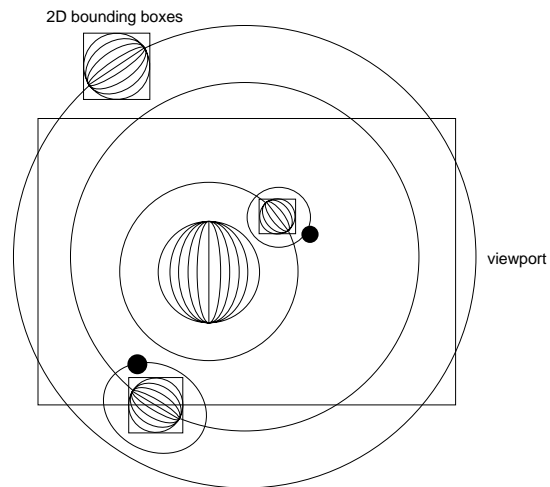


Figure 16-3 Bounding Boxes

bbox2

`bbox2()` determines whether or not an object is within the viewport, and whether it is large enough to be seen, by performing pruning and culling:

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

`bbox2()` takes as its arguments an object space bounding box ($x1, y1, x2, y2$) in object coordinates, and minimum horizontal and vertical feature sizes ($xmin, ymin$) in pixels. The system calculates the bounding box, transforms it to screen coordinates, and compares it with the viewport. If the bounding box is completely outside the viewport, the subroutines between `bbox2` and the end of the object are ignored.

If the bounding box is within the viewport, the system compares it with the minimum feature size. If it is too small in both the x and y dimensions, the rest of the subroutines in the object are ignored. Otherwise, the system continues to interpret the object.

16.2.3 Editing Objects

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

edit	add, remove, or replace drawing subroutines
tag	identify locations of drawing subroutines within an object

If you have to edit graphical objects frequently, you should build your own custom data structures and traversal subroutines, rather than use graphical objects. The editing subroutines that follow are best suited for infrequent and simple editing operations.

editobj

To open an object for editing, use `editobj()`:

```
void editobj(Object obj)
```

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

The system interprets the editing subroutines following `editobj()`. Use `closeobj()` to terminate your editing session. If you specify an undefined object, an error message appears.

getopenobj

To determine if any object is open for editing, use `getopenobj()`:

```
Object getopenobj(void)
```

If an object is open, it returns the object's id. It returns -1 if no object is open.

16.2.4 Using Tags

Tags locate items within a graphical object that you want to edit. Editing subroutines require tag names as arguments. `STARTTAG` is a predefined tag that goes before the first item in the list; it marks the beginning of the list. `STARTTAG` does not have any effect on drawing or modifying the object. Use it only to return to (find) the beginning of the list.

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

maketag

You can use tags to mark items you may want to change. You explicitly tag subroutines with `maketag()`:

```
void maketag(Tag t)
```

Specify a 31-bit numeric identifier for *t*. The system places a marker (tag) between two items. You can use the same tag name in different objects.

newtag

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

```
void newtag(Tag newtg, Tag oldtg, Offset offset)
```

istag

`istag()` tells whether a given tag is in use within the current open object:

```
Boolean istag(Tag t)
```

`istag()` returns `TRUE` if the tag is in use, and `FALSE` if it is not. The result is undefined if there is no currently open object.

gentag

`gentag()` generates a unique integer to use as a tag within the current open object:

```
Tag gentag(void)
```

deltag

`deltag()` deletes tags from the object currently open for editing:

```
void deltag(Tag t)
```

Note: You cannot delete the special tags `STARTTAG` and `ENDTAG`.

16.2.5 Inserting, Deleting, and Replacing within Objects

The subroutines `objinsert()`, `objdelete()`, and `objreplace()` allow you to add, delete, or replace subroutines in a graphical object.

objinsert

Use `objinsert()` to add subroutines to an object at the location specified in *t*:

```
void objinsert(Tag t)
```

`objinsert()` positions an editing pointer on the tag you specify in *t*. The system inserts graphics subroutines immediately after the tag. To terminate the insertion, use `closeobj()` or another editing routine (`objdelete()`, `objinsert()`, `objreplace()`).

objdelete

`objdelete()` removes subroutines from the current open object:

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

`objdelete()` removes everything between *tag1* and *tag2*, deletes subroutines and other tag names, and leaves the pointer at the end of the object after it executes. For example, `objdelete(STARTTAG, ENDTAG)` deletes every drawing routine. `objdelete()` is ignored if no object is open for editing.

objreplace

objreplace() combines the functions of objdelete() and objinsert():

```
void objreplace(Tag t)
```

This provides a quick way to replace one drawing routine with another that occupies the same amount of space in the graphical object. Its argument is a single tag, *t*. Graphics subroutines that follow objreplace() overwrite existing subroutines until a closeobj() or editing routine (objinsert(), objreplace(), objdelete()) terminates the replacement.

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

Example—Editing an Object

The following is an example of object editing. First, the object *star* is defined:

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

The object resulting from the editing session is equivalent to an object created by the following code:

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

16.2.6 Managing Object Memory

Editing can require large amounts of memory. The subroutines `compactify()` and `chunksize()` perform memory management tasks.

compactify

As memory is modified by the various editing subroutines, an open object can become fragmented and be stored inefficiently. When the amount of wasted space becomes large, the system automatically calls `compactify()` during the `closeobj()` operation.

`compactify()` allows you to perform the compaction explicitly:

```
void compactify(Object obj)
```

Unless you insert new subroutines in the middle of an object, compaction is not necessary.

Note: `compactify()` uses a significant amount of computing time. Do not call it unless the amount of available storage space is critical; use it sparingly when performance is a consideration.

chunksize

`chunksize()` lets you specify the minimum chunk of memory necessary to accommodate the largest number of vertices you want to call:

```
void chunksize(long chunk)
```

If there is a memory shortage, you can use `chunksize()` to allocate memory for an object. `chunksize()` specifies the minimum amount of memory that the system allocates to an object. The default *chunk* is 1020 bytes. When you specify *chunk*, its size should vary according to the needs of the application. As the object grows, more memory is allocated in units of size *chunk*. Call `chunksize()` only once after `winopen()`, and before the first `makeobj()`.

16.2.7 Mapping Screen Coordinates to World Coordinates

This section describes how to map screen coordinates to world coordinates.

mapw

`mapw()` takes a 2-D screen point and maps it onto a line in 3-D world space. Its argument *vobj* contains the viewing, projection, and viewport transformations that map the current displayed objects to the screen.

`mapw()` reverses these transformations and maps the screen coordinates back to world coordinates. It returns two points ($wx1, wy1, wz1$) and ($x2, wy2, wz2$), which specify two different points on the line. The length of the line is arbitrary. *sx* and *sy* specify the screen point to be mapped.

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

mapw2

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

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


Chapter 17

Feedback

This chapter describes methods used to get hardware feedback during the drawing process. Because this is a special topic with limited applications, you may want to skip this chapter on the first reading.

- Section 17.1, “Feedback on IRIS-4D/GT/GTX Systems,” describes feedback on those systems.
- Section 17.2, “Feedback on the Personal IRIS and IRIS Indigo,” describes feedback on those systems.
- Section 17.3, “Feedback on IRIS-4D/VGX, SkyWriter, and RealityEngine Systems,” describes feedback on those systems.
- Section 17.4, “Feedback Example,” demonstrates feedback.
- Section 17.5, “Additional Notes on Feedback,” discusses additional information.

Feedback is a system-dependent mechanism that uses the Geometry Pipeline to do calculations and to return the results of those calculations to the user process. From a hardware point of view, the net result of most Graphics Library calls is to send a series of commands and data down the Geometry Pipeline. In the pipeline, points are first transformed, clipped, and scaled, then lighting calculations are performed and colors are computed. Next, the points, lines, and polygons are scan-converted, and finally, pixels in the bitplanes are set to the appropriate values.

Note: Feedback is different on each IRIS-4D Series system. Avoid using the feedback mechanism unless it is absolutely necessary.

There are, however, a few places where feedback might be valuable. If you have code that draws an object on the screen, and you would like to draw the

same picture on a plotter with a different resolution than that of the screen, you can change only the `viewport()` subroutine (which controls the scaling of coordinates) so that it scales to your plotter coordinates, and then draw the picture in feedback mode. The transformed data returned to your process can often be interpreted and used to drive a plotter. `feedback()` puts the system into feedback mode, and any set of graphics subroutines can then be issued, followed by `endfeedback()`. All the commands and data that come out of the Geometry Engine subsection are as a result stored in a buffer supplied when the initial call to `feedback()` was made.

When the system is put into feedback mode, the Graphics Library commands send exactly the same information into the front of the pipeline, but the pipeline is short-circuited, and the results of some of the calculations are returned before the standard drawing process is complete. The pipeline can be broken down into many distinct stages, the first of which is composed of Geometry Engine™ processors. The Geometry Engines transform, clip, and scale vertices to screen coordinates, and do the basic lighting calculations. In feedback mode, the raw output from the Geometry Engines is sent back to the host process, and no further calculations are done.

The hardware that makes up the Geometry Engine subsection of the pipeline is different on the various IRIS workstation models. The command and data format differs and certain calculations are done on some systems and not on others. In spite of object code compatibility, the results of feedback are not compatible. If you use feedback, your code must be written differently for every system, and each time a new system is introduced, it will probably have to be modified.

Almost all feedback-type calculations can easily be done in portable host software. After a feedback session, the feedback buffer can contain any or all of the following data: points, lines, moves, draws, polygons, character move, `passthrough()`, `z-buffer`, `linestyle()`, `setpattern()`, `linewidth()`, and `lsrepeat()` values.

On the IRIS-4D/VGX, Iris Indigo, and Personal IRIS, `feedback()` returns 32-bit floating point values instead of 16-bit integers. On all other IRIS-4D Series systems, `feedback()` returns 16-bit integers.

In feedback mode, all the graphical subroutines are transformed, clipped, and scaled by the viewport, and all lighting calculations are done. Because of clipping, more or fewer vertices might appear in the feedback buffer than were sent in. A three-sided polygon can come out with up to nine sides. due to

clipping against all six clipping planes—even more side if user -defined arbitrary clipping planes are enabled (see Chapter 8).

Figure 17-1 shows the effect clipping has on feedback.

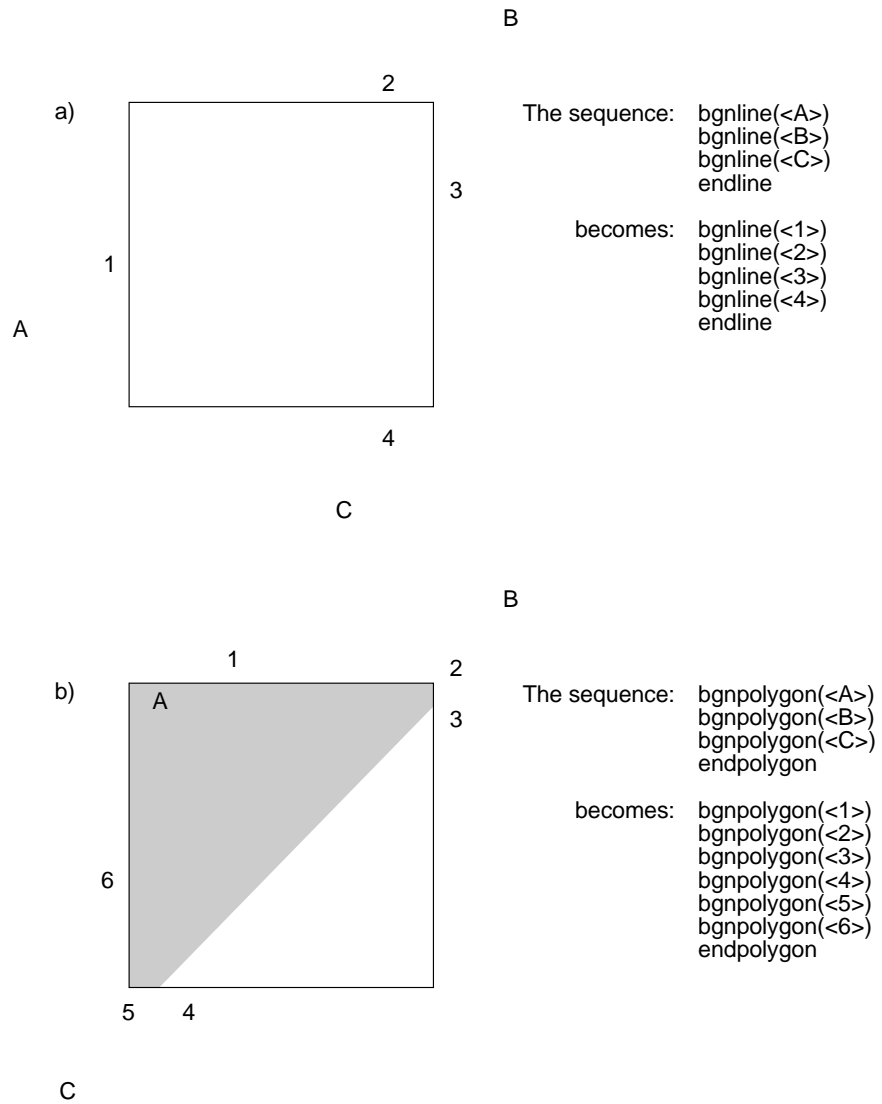


Figure 17-1 Effects of Clipping on Feedback

Because the length of the output is not generally predictable from the input, `passthrough` marks divisions in the input data. For example if you send this sequence:

```
v3f(A);
passthrough(1);
v3f(B);
passthrough(2);
v3f(C);
passthrough(3);
v3f(D);
```

the parsed information in the feedback buffer might look like this:

```
transformed point (X)
passthrough (1)
passthrough (2)
transformed point (Y)
passthrough (3)
```

Point *X* is the transformed version of point *A*, and point *Y* is the transformed version of point *C*. Points *B* and *D* must have been clipped out.

The feedback data types are in the file `gl/feed.h` for your reference. All returned information is raw and system-specific.

17.1 Feedback on IRIS-4D/GT/GTX Systems

Feedback data occurs in groups of $8n+2$ shorts, where n is the number of vertices involved, as shown in Table 17-1.

Short #	Data
1	<data type>
2	<count>
3 through (count+2)	<vertex data>

Table 17-1 IRIS-4D/G/GT/GTX Feedback Data

The vertex data is always arranged in groups of 8 (so count is a multiple of 8) and contain the values:

```
x, y, zhigh, zlow, r, g, b, alpha
```

x is the screen (not window) x-coordinate, y-1024 is the screen y-coordinate, (zhigh<<16)+zlow is the z-coordinate, and, r, g, b, and alpha are the red, green, blue, and alpha values.

By the time the data makes it through the geometry hardware, all the transformations have been done to it, including translations to put the data in the proper window. In the IRIS-4D/GT/GTX, the hardware screen y-coordinates begin at 1024 (for the bottom of the screen) and increase to 2047. Thus, 1024 must be subtracted to get what you would consider the screen y-coordinate.

24 bits of z-coordinate data is returned in two 16-bit chunks. The two chunks must be concatenated to get the full 24 bits of data. Finally, the red, green, blue, and alpha values are the colors that would be written into the frame buffer at the vertex. In RGB mode, all values vary between 0 and 255; in color map mode, the color index is sent as the red value and is in the range 0 to 4095. In color map mode, the values in the green, blue and alpha components are meaningless.

There are five possible kinds of data type: FB_POINT, FB_LINE, FB_POLYGON, FB_CMOV, and FB_PASSTHROUGH.

```
FB_POINT 24
x1, y1, zhigh1, zlow1, r1, g1, b1, alpha1
x2, y2, zhigh2, zlow2, r2, g2, b2, alpha2
x3, y3, zhigh3, zlow3, r3, g3, b3, alpha3
```

FB_LINE and FB_POLYGON are similar. FB_CMOV and FB_PASSTHROUGH always have 8 shorts of data as follows:

```
FB_CMOV 8
x1, y1, zhigh1, zlow1, r1, g1, b1, alpha1
value, junk, junk, junk, junk, junk, junk, junk
```

17.2 Feedback on the Personal IRIS and IRIS Indigo

The Personal IRIS and IRIS Indigo has the following feedback tokens defined in *gl/feed.h*:

```
FB_POINT
FB_MOVE
FB_DRAW
FB_POLYGON
FB_CMOV
FB_PASSTHROUGH
FB_ZBUFFER
FB_LINestyle
FB_SETPATTERN
FB_LINEWIDTH
FB_LSREPEAT
```

Each group of feedback data begins with one of the above tokens to indicate data type. Vertex data for points, lines, and polygons always appears in groups of six floating-point values:

```
x, y, z, r, g, b
```

x and *y* are screen (not window) coordinates, *z* is the *z* value, and *r*, *g*, *b* are the red, green, and blue (RGB) values.

The RGB values are the colors that would be written into the frame buffer at the vertex. In RGB mode, all values vary between 0 and 255. In color map mode, the *r* value is the color index (between 0 and 4095) and the *g* and *b* values are ignored.

If a move, draw, or point (as in this example) comes out of the Geometry Pipeline, the returned data consists of seven floats:

```
FB_POINT
x, y, z, r, g, b
```

For polygons, feedback data includes a count number as well as the data type number. This number indicates how many of the next float values apply to the polygon. There are six for each vertex, so this number is always a multiple of six (6, 12, etc.).

For example, the returned data for a triangle consists of 20 floats:

```
FB_POLYGON    18.0
x1, y1, z1, r1, g1, b1
x2, y2, z2, r2, g2, b2
x3, y3, z3, r3, g3, b3
```

The 18.0 indicates three vertices with six values in each; the 18 values follow.

FB_CMOV returns only three floats of data:

```
FB_CMOV
x, y, z
```

The rest of the commands (FB_PASSTHROUGH, FB_ZBUFFER, FB_LINESTYLE, FB_SETPATTERN, FB_LINEWIDTH, FB_LSREPEAT) return only one float. For example, FB_PASSTHROUGH returns:

```
FB_PASSTHROUGH
value
```

17.3 Feedback on IRIS-4D/VGX, SkyWriter, and RealityEngine Systems

The IRIS-4D/VGX and SkyWriter systems return 32-bit floating point numbers in feedback mode. The feedback data is in the following format:

```
<data type> <count> <count words of data>
```

There are five data types: FB_POINT, FB_LINE, FB_POLYGON, FB_CMOV, and FB_PASSTHROUGH. The actual values of these data types are defined in *gl/feed.h*. Following is the feedback format:

```
FB_POINT, count (9.0), x, y, z, r, g, b, a, s, t.
FB-LINE, count (18.0), x1, y1, z1, r1, g1, b1, a1, s1, t1, x2,
y2, z2, r2, g2, b2, a2, s2, t2.
FB_POLYGON, count (27.0), x1, y1, z1, r1, g1, b1, a1, s1, t1,
x2, y2, z2, r2, g2, b2, a2, s2, t2, x3, y3, z3, r3, g3, b3,
a3, s3, t3.
FB_PASSTHROUGH, count (1.0), passthrough.
FB-CMOV, count (3.0), x, y, z.
```

The *x* and *y* values are in floating point screen coordinates, the *z* value is the floating point transformed *z*. Red, green, blue, and alpha are floating point values ranging from 0.0 to 255.0 in RGB mode. In color map mode, the color

index is stored in the red value and ranges from 0.0 to 4095.0. The green, blue, and alpha values are undefined in color map mode. The *s* and *t* values are in floating point texture coordinates.

RealityEngine returns the following feedback data for points, lines, and triangles:

```
x,y,z      /* position */
r,g,b,a    /* color */
s,t,u,q    /* texture */
```

17.4 Feedback Example

The following program transforms some simple geometric figures and the results are returned in a buffer.

In this example, `feedback()` puts the system into feedback mode, and tells the system to return all data into the buffer (either *fbuf* or *sbuf*, depending on the machine type). In addition, the 110 indicates that the size of the buffer is 110 data items. If more than 110 items of data are generated, only the first 110 are saved. The geometry is drawn (in feedback mode), and `endfeedback()` ends the feedback session. `endfeedback()` returns the total number of items returned in the buffer. If an overflow occurs, the system returns an error. Finally, the loop at the end prints out the contents of the feedback buffer.

```
#include <stdio.h>
#include <string.h>
#include <gl/gl.h>

#define BUFSIZE 110

float vert[3][2] = {
    {0.1, 0.2},
    {0.7, 0.4},
    {0.2, 0.7}
};

void drawit()
{
    pushmatrix();
    color(WHITE);
    bgnpolygon();
    v2f(vert[0]);
```

```

        v2f(vert[1]);
        v2f(vert[2]);
    endpolygon();
    translate(0.1, 0.1, 0.0);
    color(RED);
    bgnline();
        v2f(vert[0]);
        v2f(vert[1]);
        v2f(vert[2]);
    endline();
    translate(0.1, 0.1, 0.0);
    color(GREEN);
    bgnpoint();
        v2f(vert[0]);
        v2f(vert[1]);
        v2f(vert[2]);
    endpoint();
    popmatrix();
}
/* feedback buffer is an array of floats on Personal Iris and
VGX */
Boolean floatfb()
{
    char model[12];
    Boolean isPI, isVGX;
    gversion(model);
    isPI = (strncmp(&model[4], "PI", 2) == 0);
    isVGX = (strncmp(&model[4], "VGX", 3) == 0);
    return (isPI || isVGX);
}

main()
{
    short sbuf[BUFSIZE];
    float fbuf[BUFSIZE];
    void *buf;
    long i, count;
    Boolean hasfloatfb;
    foreground();
    prefsiz(400, 400);
    winopen("feedback");
    color(BLACK);
    clear();
    ortho2(0.0, 1.0, 0.0, 1.0);
    hasfloatfb = floatfb();
    drawit();
}

```

```

        if (hasfloatfb)
            buf = fbuf;
    else
        buf = sbuf;
    feedback(buf, BUFSIZE);
    drawit();
    count = endfeedback(buf);
    if (count == BUFSIZE) {
        printf("Feedback buffer overflow\n");
        return 1;
    }
    else
        printf("Got %d items:\n", count);
    for (i = 0; i < count; i++) {
        if (hasfloatfb)
            printf("%.2f", fbuf[i]);
        else
            printf("%d", sbuf[i]);
        if (i % 8 == 7)
            printf("\n");
        else
            printf("\t");
    }
    printf("\n");
    sleep(10);
    gexit();
    return 0;
}

```

17.5 Additional Notes on Feedback

Any graphics subroutines can be called between `feedback()` and `endfeedback()`, but only subroutines generating points, lines, polygons, `cmovs`, or `passthroughs` can generate values in the feedback buffer. If, for example, you are writing code to generate both a display and data for a plotter, certain data can be lost (polygon patterning, for example). If it is necessary to use this information in the plotting package, you should encode it somehow into `passthrough()` commands.

Also note that subroutines such as `curve()`, `patch()`, and `mesh()`, generate feedback buffer data, because they are converted in the graphics pipeline into a series of lines or polygons.

Chapter 18

Textures

This chapter describes the texture capabilities of the IRIS GL. Texture is not available on every system. If you are not using a system that supports texture, you may want to skip to Chapter 19, “Using the GL in a Networked Environment.”

Some systems perform texturing in software, and others have special hardware for texturing. Systems that use software texturing do not exhibit the same level of texture performance as systems that use hardware texturing, so plan carefully when using texture on systems that use software texturing.

- Section 18.1, “Texture Basics,” begins by introducing some terminology that you need to know to understand textures.
- Section 18.2, “Defining a Texture,” tells you how to define a texture and how to optimize image quality.
- Section 18.3, “Using Texture Filters,” describes how to use filters to modify how textures are treated by the system.
- Section 18.4, “Using the Sharpen and DetailTexture Features,” describes two advanced texture features that are available on RealityEngine systems.
- Section 18.5, “Texture Coordinates,” tells you how to assign texture coordinates to define a mapping into object space.
- Section 18.6, “Texture Environments,” tells you how to set the texture environment to modify the color and opacity of textured polygon pixels.
- Section 18.7, “Texture Programming Hints,” presents strategies for achieving the maximum texture mapping performance from the GL.
- Section 18.8, “Sample Texture Programs,” contains two sample programs that illustrate two different ways of using textures.

18.1 Texture Basics

Texture adds realism to an image. You can use texture in a variety of ways to enhance visual information. Use texture to:

- show the material of an object. For example, wrap a wood grain pattern around a rectangular solid to create a block of wood.
- create patterned surfaces such as brick walls and fabrics by repeating textures across a surface.

See the *brick.c* sample program in Section 18.8, “Sample Texture Programs,” for an example of how to create a brick texture.

- simulate physical properties for scientific visualization applications. For example, temperature data represented by color can be mapped onto an object to show thermal gradients.

See the *heat.c* sample program in Section 18.8 for an example of how to use texture to show a thermal gradient.

- simulate lighting effects such as reflections for photo-realistic images.

Figure 18-1 shows an example of some textures that can be used to represent water, wood planks, and the end of a wood board. The water and wood plank textures are photos. The wood cross-section texture is a synthetically generated collection of light and dark hues.

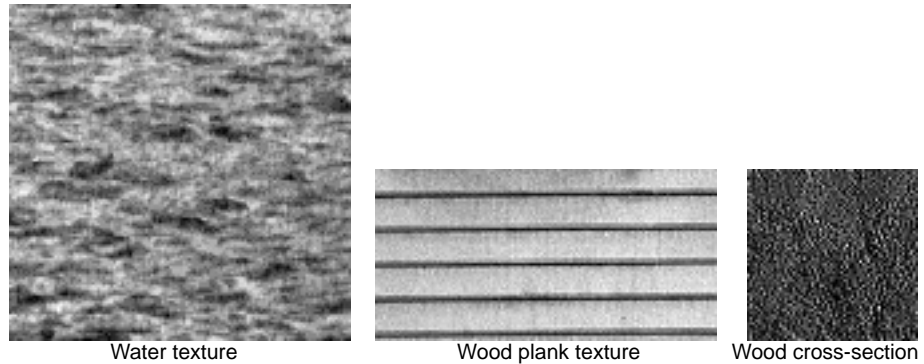


Figure 18-1 Textures for Wood and Water

Figure 18-2 shows how these textures are applied to objects in a 3-D scene.

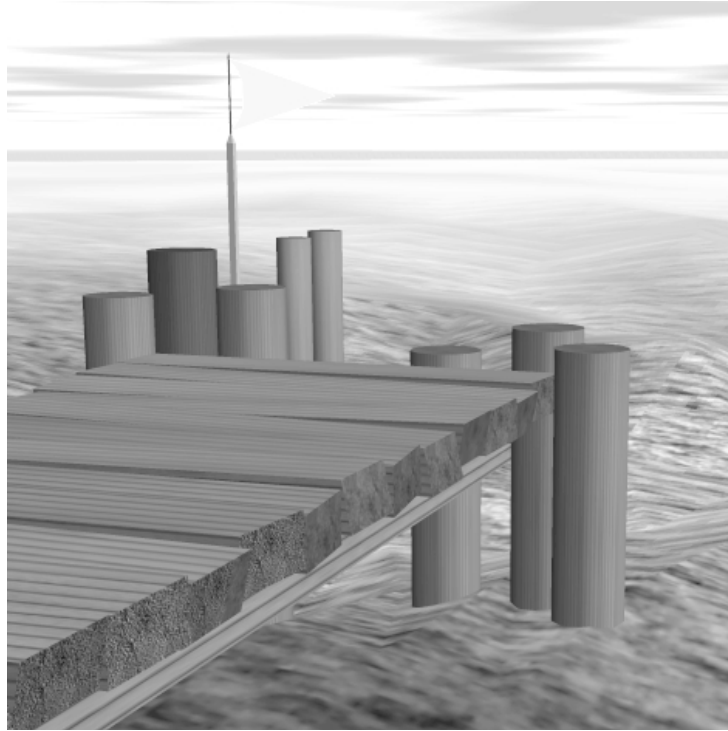


Figure 18-2 Fully Textured Scene

Textures are usually specified in two dimensions for most applications. RealityEngine systems allow three-dimensional textures. Definitions and uses of 2-D and 3-D textures follow.

18.1.1 2-D Textures

Texture mapping is a technique that applies an image to an object's surface as if the image were a decal or cellophane shrink-wrap. The image exists in a coordinate system called the *texture space*. The coordinate axes S and T define a 2-D texture space. A *texture* is a function that is defined on the interval 0 to 1 along both axes in the texture space. The individual elements of a texture are called *texels*. Texels are indexed with (s,t) coordinate pairs.

Figure 18-3 shows how a simple stripe texture is defined in 2-D texture space and mapped to 3-D object space. The mapping describes where the texels are placed in object space. This is not always a one-to-one mapping to screen pixels, as you will see later.

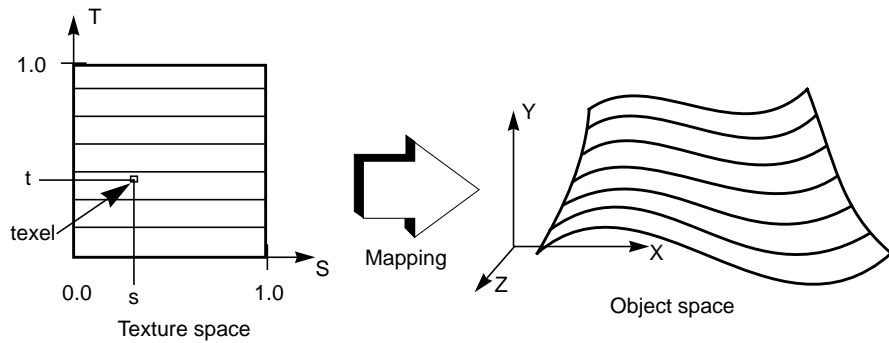


Figure 18-3 Mapping from 2-D Texture Space to 3-D Object Space

18.1.2 3-D Textures

This section describes an advanced feature that is available only on RealityEngine systems, so you may want to skip to section Section 18.1.3, “How to Set Up Texturing,” if you do not have one of these systems.

RealityEngine systems let you specify 3-D textures. Figure 18-4 shows a 3-D texture. Three-dimensional textures can also be thought of as an array of 2-D textures, as illustrated by the diagram on the right of the 3-D texture.

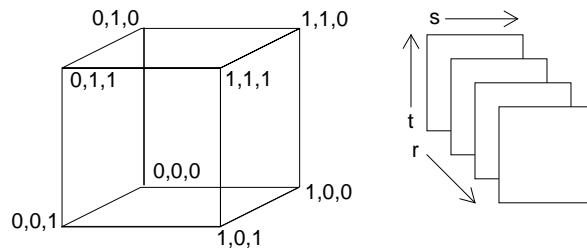


Figure 18-4 3-D Texture

The 3-D texture is mapped into (s,t,r) coordinates such that its lower left back corner is $(0,0,0)$ and its upper right front corner is $(1,1,1)$.

3-D textures can be used for

- Volume rendering.
- Examining a 3-D volume one slice at a time.
- Animating textured geometry—for example, people that move.

Texel values defined in a 3-D coordinate system form a texture volume. Textures can be extracted from this volume by intersecting it with a 3-D plane, as shown in Figure 18-5.

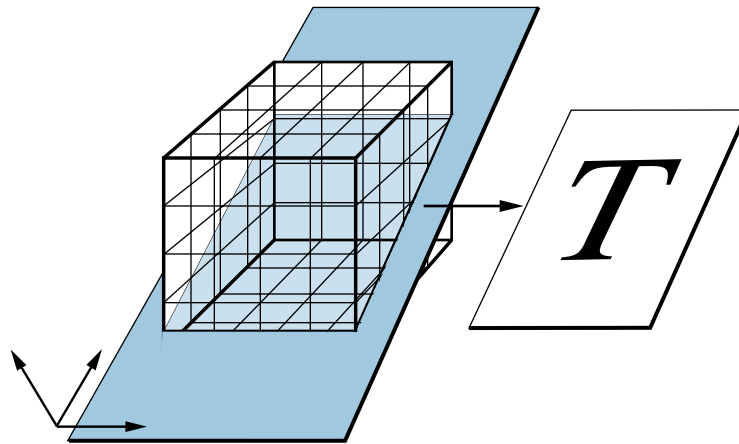


Figure 18-5 Extracting an Arbitrary Planar Texture from a 3-D Texture Volume

The resulting texture, which is applied to a polygon, is the intersection of the volume and the plane. You determine the orientation of the plane by supplying, or by having the GL supply, texture coordinates.

18.1.3 How to Set Up Texturing

This list provides an overview of the steps used to set up texturing.

1. Use `getgdesc(GD_TEXTURE)` to determine whether your system supports texturing.
2. Create a texture by defining a texture image.
3. Specify a set of texture properties that describes the number of components in the texture and how the texture should be filtered.
4. Assign texture coordinates to the vertices of geometric primitives, either explicitly or automatically, to define a mapping from texture space to geometry in object space.
5. Choose a texture environment that specifies how texture values should modify the color and opacity of an incoming shaded pixel. You can use this feature to indicate whether you want the texture to be completely opaque on top of the pixel, let some of the pixel color show through, or mix the pixel color with the texture.

Each of these steps is discussed in detail in the following sections.

18.2 Defining a Texture

A texture function consists of an image defined as an array of texels and a set of parameters that determines how samples are derived from the image. The texture image can be any image that you have constructed, scanned in, or captured from the screen.

Regardless of its dimensions, the texture image is mapped into an $(s,t,[r])$ coordinate range such that its lower-left-back corner is $(0.,0.,[0.])$ and its upper-right-front corner is $(1.,1., [1.])$.

Note: For a 2-D texture, r is always ignored.

18.2.1 Using Texture Components

The elements of the texture array are constructed with one to four components per texel. Table 18-1 lists the texture types and the components for each type.

Texture Type	Components
1-component	Intensity
2-component	Intensity, Alpha
3-component	Red, Green, Blue
4-component	Red, Green, Blue, Alpha

Table 18-1 Texture Components

Intensity is used to show color variations (shades) within the same color value. Alpha is used to indicate the transparency of the color.

Suggestions for choosing a texture type to achieve certain effects follow.

Use a 1-component texture to create subtle variations in surfaces. For example, vary the intensity of a brown shade to turn plain brown hills into hills with shades of light to dark brown. You can also use a 1-component texture with a texture environment to create blended textures, such as a blue-and-white sky. The wood and the water in Figure 18-2 are examples of 1-component textures.

Use a 2-component texture to create surfaces with subtle color variations on geometry that has irregular edges. For example, use a 2-component texture to create a tree with many shades of green. The 2-component texture used for the foliage varies in intensity, creating different shades of green from light to dark. Another 2-component texture is used for the trunk, which has different shades of brown. 1-component and 2-component textures are sufficient for representing many types of objects and are effective because they use less memory than 3- and 4-component textures.

Alpha, the other component of the 2-component texture, is used to indicate how transparent the texture is. Commands that determine how alpha affects the manner in which a texture is drawn include `afunction()`, `blendfunction()`, and on Reality Engine, `msalpha()`.

Note: On RealityEngine, use multisampling with the `msalpha()` feature rather than `afunction()` to define the edges of the tree.

Figure 18-6 shows an example of a tree created with a 2-component texture. The tree is a single rectangular polygon that has a scanned photo of a tree superimposed on it. This polygon can be rotated about the center of the trunk, as shown by the outlines, so that it is always facing the viewer.

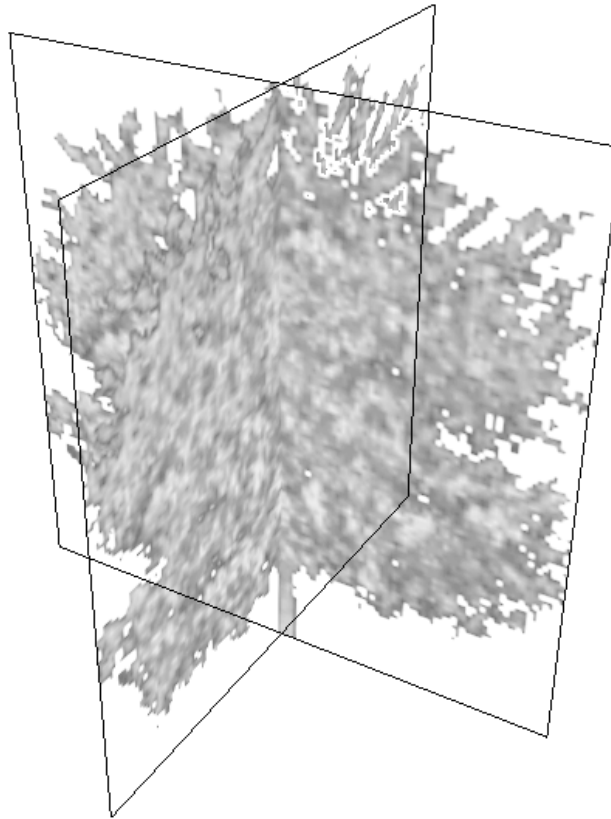


Figure 18-6 Example of a Tree Created with a 2-component Texture

Representing complex surfaces by texturing simple polygons rather than by creating complex geometry with multiple polygonal faces can achieve greater realism and better performance. You can experiment with the performance trade-off between the number of polygons and the use of texturing to get the best possible solution for your application.

18.2.2 Loading a Texture Array

Textures are loaded into a memory array that the system accesses when rendering the textured surface. Figure 18-7 shows how the texture array is constructed for an 8-bit-per-component image.

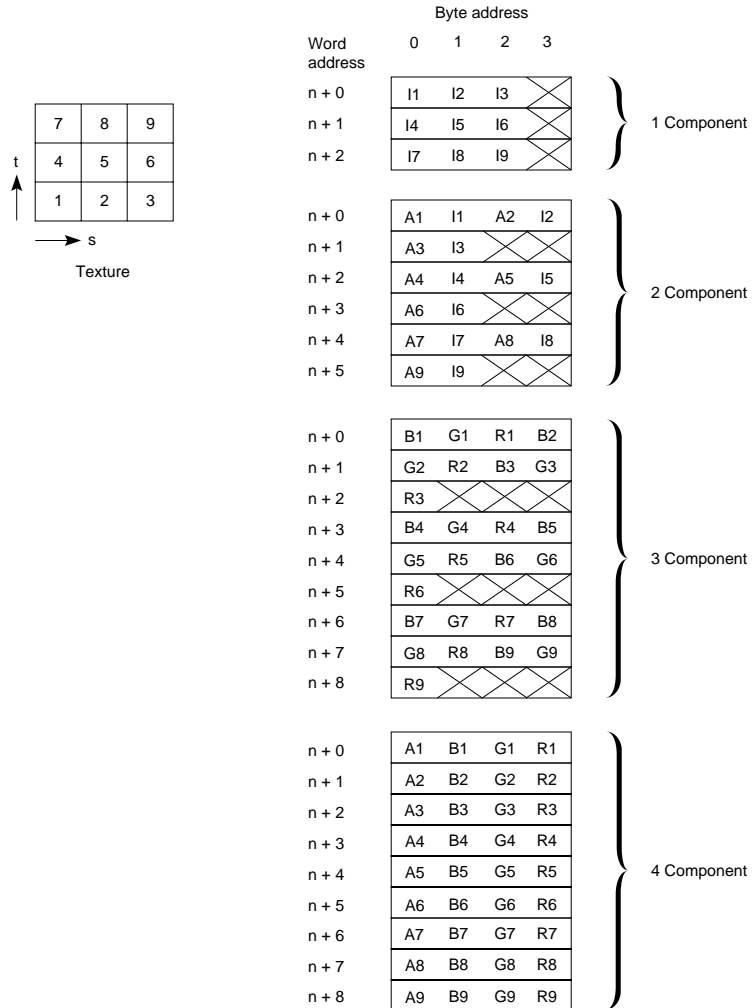


Figure 18-7 Structure of a Texture Array

Figure 18-7 shows a texture consisting of 9 texels, which are numbered 1 through 9. The texels fill the texture from left to right, bottom to top. The component information for each texel is stored as a packed array of unsigned long words. This is the same format used by `lrectread()`.

In Figure 18-7, the boxes represent blocks of memory. A long word is 32 bits, and each byte of texture information requires 8 bits. Therefore, 4 bytes of texel information can fit into each long word. Each row of texel information must be long word-aligned, so the end of the row must be byte-padded to the end of each long word. The diagram shows how the array is packed for an 8-bit-per-component texture of 1-, 2-, 3-, or 4- components, consisting of 9 texels.

Table 18-2 summarizes the relationships between texel component and byte ordering.

Components	Pixel Type	Byte Ordering (low-order to high-order)
1-Component	Intensity	I0, I1, I2, I3, I4,...
2-Component	Intensity-Alpha	A0, I0, A1, I1, A2,...
3-Component	Red, Green, Blue	B0, G0, R0, B1, G1,...
4-Component	Red, Green, Blue, Alpha	A0, B0, G0, R0, A1,...

Table 18-2 Texture Image Array Format

For each polygon pixel to be textured, the texture function generates texture components (color, intensity, alpha) based on the texel type, the texture map coordinates of the pixel's center, and the area in texels onto which the pixel maps. The properties that you specify for the texture function determine how the texture image is sampled and how the texture function is evaluated outside the range (0.,0.,[0.]), (1.,1.,[1.]).

Note: All geometry including polygons, lines, points, and character strings are texture-mapped. Character strings always have the texture coordinates (0.,0.,[0.]).

18.2.3 Defining and Binding a Texture

Textures use the define/bind paradigm that was introduced in Chapter 9.

Use `texdef2d()` or `texdef3d()` to *define* a texture and `texbind()` to *activate* a texture.

Textures can be redefined by calling `texdef2d()` or `texdef3d()` with the index of a previously defined texture. As with materials, only one texture can be active, or bound, at a time. The binding process and defining process are separated for performance reasons—it takes substantially less time to bind a texture than it takes to define one.

The ANSI C specifications for `texdef2d()` and `texdef3d()` are:

```
void texdef2d(long index, long nc, long width, long height,  
             unsigned long *image, long np, float props)
```

```
void texdef3d(long index, long nc, long width, long height,  
             long depth, unsigned long *image, long np, float props)
```

where:

<i>index</i>	is a unique index, or name, that identifies the texture. Index 0 is reserved as a null definition, and it cannot be redefined.
<i>nc</i>	is the number of components per texel (1, 2, 3, or 4).
<i>width</i>	is the width of the texture image in texels.
<i>height</i>	is the height of the texture image in texels.
<i>depth</i>	is the depth of the texture image in texels.
<i>image</i>	is a word-aligned array containing the texel data.
<i>np</i>	is the number of symbols and floating point values in the <i>props</i> array, including the termination symbol <code>TX_NULL</code> . If <i>np</i> is zero, it is ignored, but operations over network connections are more efficient when <i>np</i> is correctly specified.
<i>props</i>	is an array of floating point symbols and values that define how to interpret the texture function. The <i>props</i> array contains a sequence of symbols, each followed by the appropriate number of floating point values. The last symbol in the array must be <code>TX_NULL</code> , which terminates the array.

The following code fragment illustrates how to use `texdef2d()` to define a 2-D brick texture:

```
float texprops[] = {TX_MINFILTER, TX_POINT,  
                   TX_MAGFILTER, TX_POINT,  
                   TX_WRAP, TX_REPEAT, TX_NULL};  
  
texdef2d(1, 1, 8, 8, bricks, 7, texprops);
```

The current texture, `bricks` in this case, is bound using the `texbind()` call:

```
texbind(TX_TEXTURE_0, 1);
```

In this example, the array `texprops` explicitly specifies `TX_MINFILTER`, the filter function that is used for minifying texture when the pixel being textured maps onto an area greater than one texel, and `TX_MAGFILTER`, the filter function that is used when the pixel being textured maps to an area less than or equal to one texel. See Section 18.3, “Using Texture Filters,” for a discussion of minification and magnification filters. In the brick example, `TX_MINFILTER` and `TX_MAGFILTER` are both set to use point-sampling filters.

`TX_WRAP`, which specifies what to do when the $(s, t, [r])$ coordinates are outside the range 0.0 through 1.0, is set to `TX_REPEAT`. `TX_REPEAT` specifies that only the fractional parts of the texture coordinates are used, thereby creating a repeating pattern. `TX_REPEAT` is the default. By setting `TX_WRAP` to `TX_REPEAT`, the small 8×8 pattern is repeated across the polygon, creating an entire wall of bricks.

You can specify the wrapping behavior per coordinate, rather than globally:

`TX_WRAP_S` specifies the wrapping behavior only for the *s* texture coordinate.

`TX_WRAP_T` specifies the wrapping behavior only for the *t* texture coordinate.

`TX_WRAP_R` specifies the wrapping behavior only for the *r* texture coordinate.

If you replace `TX_REPEAT` with `TX_CLAMP`, you see the brick pattern only once on the polygon, where the (s, t) coordinates are in the range $(0, 1)$. The edges of the texture are smeared across the rest of the polygon. `TX_CLAMP` is useful for preventing wrapping artifacts when mapping a single image onto an object.

`TX_TILE`, a property that is not used in the brick example, supports mapping of high-resolution images with multiple rendering passes. By splitting the texture into multiple pieces, each piece can be rendered at the maximum supported texture resolution. For example, to render a scene with 2× texture

resolution, `texdef2d()` is called four times. Each call includes the entire image, but specifies a different subregion of that image to be converted into a texture.

`TX_TILE` is followed by four floating point coordinates that specify the x and y coordinates of the lower-left corner of the subregion, then the x and y coordinates of the upper-right corner of the subregion. The original texture image continues to be addressed in the range 0,0 through 1,1. However, the subregion occupies only a fraction of this space, and pixels that map outside the subregion are not drawn.

To divide the image both horizontally and vertically into quadrants, the corners of the subregions should be (0,0 .5,.5), (.5,0 1,.5), (0,.5 .5,1), and (.5,.5 1,1). The scene is then drawn four times, each time calling `texbind()` with the texture ID of one of the four quadrants. In each pass, only the pixels whose texture coordinates map within that quadrant are drawn.

If the image, or the specified subregion of the image, is larger than what can be handled by the hardware, it is reduced to the maximum supported size automatically, with no indication other than the resulting visual quality. Because subregions are specified independently, they should all be the same size. Otherwise, some subregions may be reduced while others are not.

18.2.4 Selecting the Texel Size

This section describes an advanced feature that is available only on RealityEngine systems, so you may want to skip to Section 18.3, “Using Texture Filters,” if you do not have one of these systems.

RealityEngine supports three internal texel sizes: 16-bit, 32-bit, and 64-bit. You can change this internal format to select the texel size that best suits your application needs. There is a trade-off between image quality and speed. The fill rate is inversely proportional to the texel size; thus, the fill rate doubles when the texel size is halved.

The default texel size for 1- and 2-component textures is 16 bits. The default texel size for 3- and 4-component textures is 32 bits.

Each of the texel sizes is available with 12, 8, or 4 bits per component. Table 18-3 shows the configurations possible, and the symbols for selecting those configurations for the different texel sizes.

Texel Size	1-component	2-component	3-component	4-component
16-bit	TX_I12_A4	TX_I12_A4, TX_IA_8	TX_RGB_5	TX_RGBA_4
32-bit		TX_IA_12	TX_RGBA_8	TX_RGBA_8
64-bit			TX_RGB_12	TX_RGBA_12

Table 18-3 Texture Component Configuration for Different Texel Sizes

Use 16-bit texels for the fastest performance and to reduce memory usage. 16-bit texels with `TX_RGBA_4` provide high performance and good image quality, but if you don't need alpha, use `TX_RGB_5` for even better image quality, because it increases the color resolution.

Use the 64-bit texel size for the highest resolution for color computations, for example, in low-light-level simulations. This format provides 12-bit per component capability for R,G,B,A texture maps. The advantage of 12 bits per component is that it increases the number of color levels for each component from 256 to 4096, greatly enhancing the precision of the color computation.

Use the 32-bit texel size when you want to balance performance halfway between speed and image quality.

The texel size and bit configuration of the texture components are set as internal and external format hints in the *props* array of the `texdef2d()` and `texdef3d()` commands.

Use `TX_INTERNAL_FORMAT` in the *props* array as a hint to trade image quality for speed. This hint affects the precision used internally in texture function computations. Because the performance of texture function implementations is typically constrained by texel accesses per screen pixel, you can specify a smaller internal texel size and often realize performance gain.

The tokens for `TX_INTERNAL_FORMAT` are:

- `TX_I_12A_4` specifies that a 1- or 2-component texture should be computed with at least 12 bits for intensity and 4 bits for alpha. Texel size: 16 bits.
- `TX_IA_8` specifies that a 2-component texture should be computed with at least 8 bits for intensity and 8 bits for alpha. Texel size: 16 bits.
- `TX_RGB_5` specifies that a 3-component texture should be computed with at least 5 bits for red and blue and at least 6 bits for green. Texel size: 16 bits.
- `TX_RGBA_4` specifies that a 4-component texture should be computed with at least 4 bits per component. texel size: 16 bits.
- `TX_IA_12` specifies that a 2-component texture should be computed with at least 12 bits per component. Texel size: 24 bits; may be rounded up to 32 bits.
- `TX_RGBA_8` specifies that a 3- or 4-component texture should be computed with at least 8 bits per component. Texel size: 32 bits.
- `TX_RGBA_12` specifies that a 4-component texture should be computed with at least 12 bits per component. Texel size: 64 bits.
- `TX_RGB_12` specifies that a 3-component texture should be computed with at least 12 bits per component. Texel size: 48 bits, rounded to 64 bits.

`TX_EXTERNAL_FORMAT` specifies the size of the image components:

- `TX_PACK_8` specifies that the image is composed of 8-bit components. This is the default.
- `TX_PACK_16` specifies that the image is composed of 16-bit components.

When the external format is larger than the internal format, the most significant bits of the external format pixel are used. When the external format is smaller than the internal format, the most significant bits of the external format pixel are replicated in the lower order bits of the internal format. Thus, three 8-bit external format components with the hexadecimal values `AB,FF,00` become the three 12-bit internal format components with the hexadecimal values `ABA,FFF,000`.

The next section describes the filters that can be specified in the *props* array.

18.3 Using Texture Filters

During the texture mapping process, the texture function computes texture values based on the $(s, t, [r])$ texture coordinates at the center of the polygon pixel that is being textured and the area in texture space onto which the pixel maps. One of two filtering algorithms is used, depending on the size of this area.

If the area is greater than the area of 1 texel, as shown in Figure 18-8, the texture is *minified* to fit the screen pixel and the texture function's minification filter is used. Specify the minification filter with the `TX_MINFILTER` parameter.

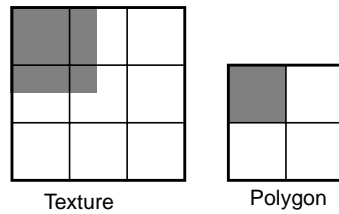


Figure 18-8 Texture Minification

If the area is less than the area of 1 texel, as shown in Figure 18-9, the texture is *magnified* to fill the screen pixel and the texture function's magnification algorithm is used. Specify the magnification filter with the `TX_MAGFILTER` parameter.

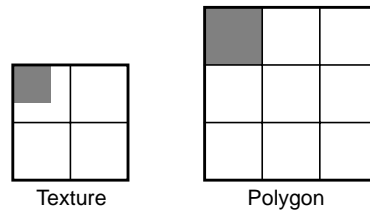


Figure 18-9 Texture Magnification

Minification and magnification filters are discussed in detail in the sections that follow.

18.3.1 Minification Filters

Minification filters are used when multiple texels correspond to a single screen pixel, as shown in Figure 18-10.

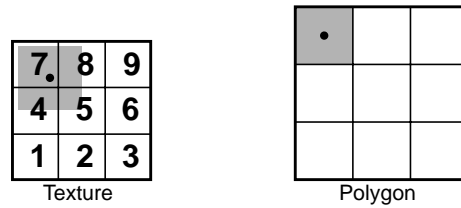


Figure 18-10 Texture Minification

In most cases, the best minification results are obtained by using a MIPmap to minify the texture.

MIPmap Minification Filters

Figure 18-11 shows a MIPmap. MIP comes from a Latin term that means “many things in a small place.” A MIPmap stores an array of prefiltered versions of the texture image.

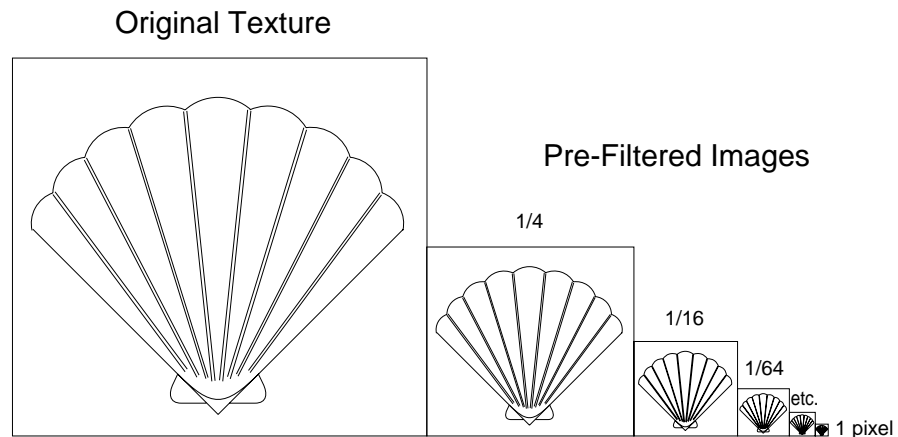


Figure 18-11 MIPmap

Each image in the array has half the resolution of the image before it, but it still maps into the texture coordinate range (0.,0.) to (1.,1.). Thus, the first image in the MIPmap has a 1-to-1 texel-to-pixel correspondence. The second image has a 4-to-1 correspondence, the third image, 16-to-1, and so on.

For any minification factor, there is one image in the MIPmap whose texels map closely to an area in texture space that is less than or equal to the area that the pixel being textured maps into. This image has the appropriate resolution, so samples interpolated from this image do not have undersampling artifacts.

Each of the MIPmap filters works differently. The default minification filter for systems other than RealityEngine is `TX_MIPMAP_LINEAR` or a filter of equal performance, but better quality. Prefiltered versions of the image, when required by the minification filter, are computed automatically by the GL.

RealityEngine uses high-performance trilinear MIPmap filtering by default. Simultaneous parallel memory access allows the eight samples needed for trilinear interpolation to be retrieved with a single memory access.

Trilinear interpolation is one of the highest quality texture functions available. It produces images that look sharp when viewed from close range and that remain stable under all circumstances. In addition, there is no perceptible transition in the image as the textures move relative to the eyepoint.

RealityEngine also performs *quadlinear* MIPmap filtering of 3-D textures. This is effectively a trilinear interpolation of a 3-D texture, automatically generating a series of 3-D volumes, each 1/8 smaller than the one above. The interpolation is performed between the 8 adjacent pixels in the MIPmap from the two closest-bounding volume levels and then blended between the two results, thus achieving a four-way interpolation.

Select the filter to use based on the type of application you are creating and the quality and performance results you want. Refer to Section 18.7, "Texture Programming Hints," for additional information on selecting a filter.

Note: Because the high-performance MIPmap filters available on RealityEngine are superior to other MIPmap minification filters, the GL always uses `TX_MIPMAP_TRILINEAR` for MIPmapping 2-D textures and `TX_MIPMAP_QUADLINEAR` for MIPmapping 3-D textures for applications running on a RealityEngine, no matter what filter is specified in the props array.

To select a minification filter, use the token `TX_MINFILTER`, followed by a single symbol that specifies the minification filter. Values for `TX_MINFILTER` are listed below, with descriptions of what they do.

Note: Filters marked with an asterisk(*) are currently available only on RealityEngine systems.

`TX_MIPMAP_POINT`

chooses a prefiltered version of a 2-D texture, based on the number of texels that correspond to 1 screen pixel. The value of the pixel that is nearest to the (s,t,r) mapping onto that image is used to color the pixel.

`TX_MIPMAP_LINEAR`

chooses the two prefiltered versions of a 2-D texture that have the nearest texel-to-screen pixel correspondence. A weighted average of the values of the pixel in each of these images that is nearest to the (s,t,r) mapping onto that image is used to color the pixel.

`TX_MIPMAP_BILINEAR`

chooses a prefiltered version of a 2-D texture, based on the number of texels that correspond to 1 screen pixel. The weighted average of the values of the 4 pixels nearest to the (s,t) mapping onto that image is used to color the pixel.

`TX_MIPMAP_TRILINEAR`

chooses the prefiltered version of the 2-D texture whose texel size most closely corresponds to screen pixel size. A weighted average of the values of the pixels nearest to the mapping onto that image is used to color the pixel.

For 2-D textures, `TX_MIPMAP_TRILINEAR` chooses the two prefiltered versions of the image that have the nearest texel-to-screen pixel size correspondence. A weighted average of the values of the 4 pixels in each of these images that are nearest to the (s,t) mapping onto that image is computed. The weighted averages from the two levels are then themselves interpolated.

For 3-D textures, this filter is analogous to `MIPMAP_BILINEAR` for the 2-D textures—that is, the filter chooses the prefiltered MIPmap image whose texel size most closely corresponds to screen pixel size and uses the weighted average of the values of the 8 pixels nearest to the (s,t,r) mapping onto that image.

Note: TX_MIPMAP_TRILINEAR is available only on SkyWriter, VGXT, and RealityEngine systems.

TX_MIPMAP_QUADLINEAR*

chooses the two prefiltered versions of a 3-D texture that have the nearest texel-to-screen pixel size correspondence. A weighted average of the 8 pixels in each of these images that are nearest to the (s,t,r) mapping onto that image is computed. The weighted averages from the two levels are then themselves interpolated.

TX_MIPMAP_FILTER_KERNEL*

specifies an 8x8x8 kernel to use as a separable symmetric filter to generate MIPmap levels. Because it is separable and symmetric, only one dimension needs to be specified. The eight floating point values that follow the token specify the kernel. The default that is used for implementations which do not correct for perspective distortion is 0.0, 0.0, 0.125, 0.375, 0.375, 0.125, 0.0, 0.0. The default that is used for implementations which correct for perspective distortion is 0.0, -0.03125, 0.05, 0.48125, 0.48125, 0.05, -0.03125, 0.0. This filter blurs less than the others.

Other Minification Filters

Minification can be performed without MIPmapping. To minify textures without using a MIPmap, select one of these filters:

Note: Filters marked with an asterisk(*) are currently available only on RealityEngine systems.

TX_POINT uses the value of the texel, in either a 2-D or 3-D texture, that is nearest to the (s,t,r) mapping onto the texture to color the pixel.

TX_BILINEAR uses a weighted average of the values of the 4 texels in a 2-D texture that are nearest to the (s,t) mapping onto the texture.

TX_TRILINEAR* uses a weighted average of the values of the 8 texels of a 3-D texture that are nearest to the (s,t,r) mapping onto the texture.

TX_BICUBIC* computes a smoothly weighted average of a 4x4 region of texels in a 2-D texture that are nearest to the (s,t) mapping onto the texture.

The drawback of using either the `TX_POINT` or the `TX_BILINEAR` filter for minification is that only 1, or 4, of the texture pixels that map onto the area of the pixel being textured are considered in the texture value computation. If the texture is mapped so that it is shrunk by a factor greater than two, it may exhibit *scintillation*, a shimmering or swimming motion as if it is not tacked firmly to the surface, or it may appear to have a *moire* pattern on top of it.

Aliasing artifacts such as these result from *undersampling*—not including in the texture value computation the contributions of all of the texture pixels that map onto the pixel being textured. Artifacts caused by undersampling can be alleviated by using one of the MIPmap filters.

To see how MIPmap filtering reduces aliasing and blockiness, change the `texprops` array of the brick texture to:

```
float texprops[] = {TX_MINFILTER, TX_MIPMAP_BILINEAR,
TX_MGFILTER, TX_BILINEAR, TX_WRAP, TX_REPEAT, TX_NULL};
```

Sometimes you may not want the blurring that results from MIPmap filtering, as is frequently the case when texture alpha is used as a geometry approximating template—for example, in defining the outline of a row of trees. In these circumstances, `TX_BILINEAR` is a good minification filter choice on systems other than RealityEngine. RealityEngine supports a feature called `SharpenTexture`, described in Section 18.4, “Using the Sharpen and DetailTexture Features,” to maintain the crispness of edges on textured geometry.

18.3.2 Using Magnification Filters

Magnification filters are used when multiple screen pixels correspond to 1 texel, as shown in Figure 18-12.

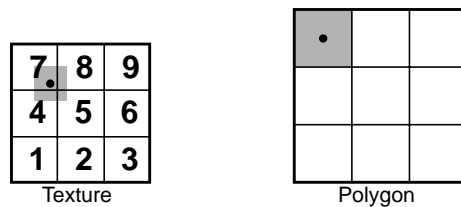


Figure 18-12 Texture Magnification

To select a magnification filter, use the token `TX_MAGFILTER`, followed by a single symbol that specifies the magnification filter. Values for `TX_MAGFILTER` are listed below, with descriptions of what they do.

Note: Filters marked with an asterisk (*) are currently available only on RealityEngine systems.

- `TX_POINT` Used for either 2-D or 3-D textures to select the value of the texel nearest to the $(s,t,[r])$ mapping onto the screen pixel of the polygon that is being textured. For example, in Figure 18-12, `TX_POINT` selects texel number 7 for texturing the highlighted polygon pixel.
- On systems other than RealityEngine, `TX_POINT` is generally faster than `TX_BILINEAR`, but has the drawback that mapped textures can appear boxy because there is not as smooth a transition between the texels as there is with `TX_BILINEAR`. If the texture image does not have sharp edges, this effect might not be noticeable.
- `TX_BILINEAR` Used for 2-D textures, to select the weighted average of the values of the 4 texels nearest to the (s,t) mapping onto the texture. For example, in Figure 18-12, `TX_BILINEAR` would cause a weighted average of texels 4, 5, 7, and 8 to be used to color the screen pixel.
- `TX_TRILINEAR*` Used for 3-D textures, to select the weighted average of the values of the 8 texels nearest to the (s,t,r) mapping onto the texture.
- `TX_BICUBIC*` Used for 2-D textures, to compute a smooth weighted average of a 4×4 region of texels nearest to the (s,t) mapping onto the texture.

See the *texdef(3G)* man page for the formulas used to compute filter parameters.

See Section 18.4, “Using the Sharpen and DetailTexture Features,” for information on three additional magnification filters—`TX_SHARPEN`, `TX_ADD_DETAIL`, and `TX_MODULATE_DETAIL`—that can be used for enhancing the image quality of magnified textures on RealityEngine systems.

18.4 Using the Sharpen and DetailTexture Features

This section describes an advanced feature that is available only on RealityEngine systems, so you may want to skip to Section 18.5, “Texture Coordinates,” if you do not have one of these systems.

The appearance of a textured surface can vary, depending on whether it is seen from a distance or close up. For example, from a distance you see the lane markings and reflectors on a road, but close to its surface you see only gravel and tar.

There are two types of problems that occur when the eyepoint is close to a textured surface:

- The texture lacks sufficient detail for close-ups.
- The texture image is out of focus as a result of over-magnification.

RealityEngine provides solutions for these problems with Sharpen and DetailTexture. These two features enable low-resolution textures to be as crisp as high-resolution textures without taking up a lot of texture storage space.

Sharpen works best when the high-frequency information is used to represent edge information. A stop sign is an example of this type of texture—the edges of the letters have distinct outlines. Magnification normally causes the letters to blur, but Sharpen keeps the edges crisp.

DetailTexture works best for a texture with high-frequency information that is not strongly correlated to its low-frequency information. This occurs in images that have a uniform color and texture variation throughout, such as a field of grass or a wood panel with a uniform grain.

18.4.1 Using the Sharpen Feature

Textures must often be magnified for close-up views. However, not all textures can be magnified without looking blurry or artificial. The fine details of a texture, such as the precise edges of letters on a sign, are supplied by high-frequency image data within a high-resolution image. When the high-frequency data is missing, the image is blurred.

Sharpen uses the top two levels of a MIPmap to *extrapolate* high-frequency information *beyond* the texture image in the top level of the MIPmap.

Sharpen lets you use a lower resolution texture map, yet preserve the sharpness of the edges in the original image. This allows you to use less texture storage per texture.

Sharpen maintains edges that bilinear magnification normally blurs. For example, Sharpen works exceptionally well for textures such as the stop sign and for textures whose alpha represents geometry with intricate edges, such as a tree. During the magnification process the edges are extrapolated and they stay crisp.

To use Sharpen, specify the `TX_SHARPEN` token for `TX_MAGFILTER`.

How Sharpen is Computed

The GL computes a Level-of-Detail (LOD) factor at each pixel it textures. LOD is the magnification factor above the base level. LOD n is a 2^n magnification. For example, if a 512×512 base texture is LOD 0, its LOD (-1) texture is 256×256 .

To produce a sharpened texel n LODs above the base texture, the GL adds n times the weighted difference between the texel at LOD 0 and LOD (-1) to LOD 0, or

$$\text{LOD}_n = \text{LOD}_0 + \text{weight}(n) * (\text{LOD}_0 - \text{LOD}(-1))$$

where:

n is the number of levels of extrapolation.

$\text{weight}(n)$ is the sharpening multiplier function.

LOD 0 is the base texture.

LOD (-1) is the texture at half resolution.

By default, the GL uses a linear extrapolation function, where $\text{weight}(n) = n$.

Customizing the Sharpen Function

Sharpen can cause ringing in some textures when they are magnified too much. The weight can be varied to create a nonlinear LOD extrapolation curve and/or the extrapolation function can be clamped to reduce the ringing.

Figure 18-13 shows LOD extrapolation curves as a function of weight and magnification factors.

The curve on the left is the default linear extrapolation, where $\text{weight}(n)=1*n$. The curve on the right is a nonlinear extrapolation, where the weight function is modified to control the amount of sharpening so that less sharpening is applied as the magnification factor increases.

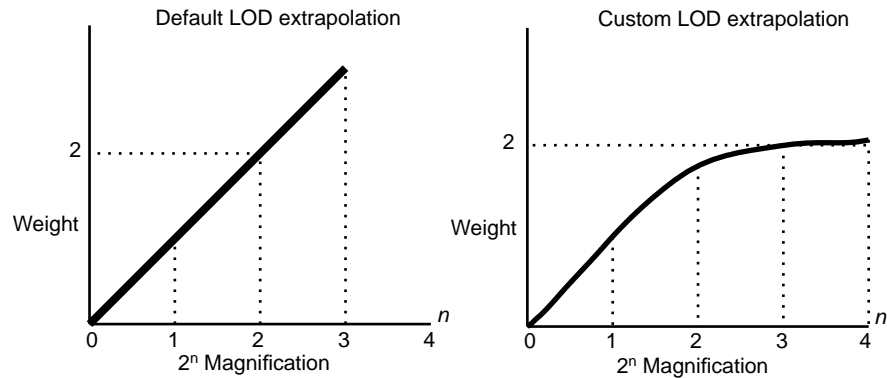


Figure 18-13 LOD Extrapolation Curves

Use `TX_CONTROL_POINT` to specify control points for shaping the sharpen function. The first control point specifies the LOD, and the second control point specifies a weight multiplier for that magnification level.

For example, to gradually ease the sharpening effect—use a nonlinear LOD extrapolation curve, as shown on the right in Figure 18-13—with these control points:

```
TX_CONTROL_POINT, 0., 0.,  
TX_CONTROL_POINT, 1., 1.,  
TX_CONTROL_POINT, 2., 1.7,  
TX_CONTROL_POINT, 4., 2.0,
```

If a texture exhibits ringing when it is magnified with Sharpen—for example, beyond a 6× magnification, you can set the `TX_CONTROL_CLAMP` to clamp at the maximum allowable extrapolation.

Figure 18-14 shows how the default linear extrapolation on the left can be clamped at an arbitrary LOD value, 2.5 in this case, beyond which extrapolation is clamped.

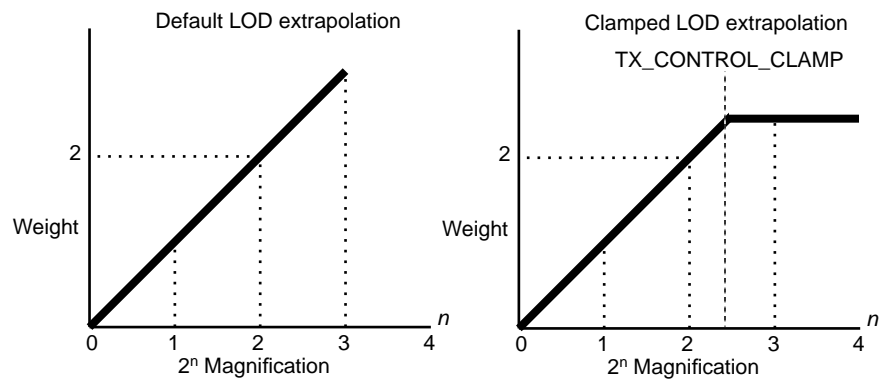


Figure 18-14 Clamping the LOD Extrapolation

Specify a clamp at LOD 2.5 as follows:

```
TX_CONTROL_CLAMP, 2.5
```

You can sharpen the alpha or the color of a texture independently by explicitly setting the magnification filter to use for color and alpha. For example, use the following functions to maintain the precise edges of a geometry described by alpha such as a tree, while allowing the colors to blur:

```
TX_MAGFILTER_ALPHA, TX_SHARPEN,  
TX_MAGFILTER_COLOR, TX_BILINEAR,
```

18.4.2 Using DetailTextures

Ideally, you would always use textures that have high enough resolution to allow magnification without bluriness. High-resolution textures maintain realistic image quality for both close-up and distant views. For example, in a high-resolution road texture, both the large features, such as potholes, oil stains, and lane markers that are visible from a distance, as well as the asphalt of the road surface, look realistic no matter where the viewpoint is.

Unfortunately, a high-resolution road texture with that much detail may be as large as 2K×2K, which exceeds the maximum texture storage capacity of the system. Making the image close to or equal to the maximum allowable size still leaves little or no memory for the other textures in the scene.

RealityEngine provides a solution for representing the 2K×2K road texture with the DetailTexture feature.

How DetailTexture Works

The detail elements of a texture, such as the asphalt in a road texture, are the high-frequency components of a high-resolution image. Because the high-frequency detail is virtually the same across a texture such as a road, the high-frequency detail from any portion of the image can be used as the high-frequency detail across the entire image.

Using the same high-frequency detail across the entire image allows the high-resolution image to be represented with the combination of a low-resolution image and a small high-frequency detail image, which is called a DetailTexture. RealityEngine can combine these two images on-the-fly to create an approximation of the high-resolution image.

Creating a DetailTexture and a Low-Resolution Texture

You can convert a high-resolution image into a low-resolution image and a DetailTexture in the following manner:

Make the low-resolution image by shrinking the high-resolution image to the desired resolution. You can then extract the high-frequency detail from the high-resolution image by scaling the low-resolution image back up to the size of the high-resolution image, then subtracting it from the original high-resolution image.

The result is a *difference image* that contains only the high-frequency details of the image. You can use any 256×256 subimage of this difference image as a `DetailTexture`.

For example, follow these steps to create a 512×512 low-resolution texture, and a `DetailTexture` from a 2K×2K high-resolution image:

1. Make the low-resolution image as follows:

Use *izoom* or other resampling program to make the low-resolution image by shrinking the high-resolution image by 2^n . In this example, n is 2, so the resolution of the low-resolution image is 512×512. This band-limited image has had the n highest frequency bands of the original image removed from it.

2. Make the `DetailTexture` as follows:

1. Use *subimage*, or other tool to select a 256×256 region of the original high-resolution image, 2K×2K in this case, whose n highest frequency bands are characteristic of the image as a whole.

For example, rather than choosing a subimage from the lane markings, choose an area in the middle of a lane.

2. Optionally, you can make this image self-repeating along its edges to eliminate the seams.
3. Make a blurry version of this 256×256 subimage.

First, shrink the 256×256 subimage by 2^n , to 64×64 in this case.

Now, scale the resulting image back up to 256×256.

This image is blurry because it is missing the two highest frequency bands present in the two highest levels of detail (LOD).

4. Subtract the blurry subimage from the original subimage. This signed difference image has only the 2 highest frequency bands.
5. Add a bias to make the image unsigned. If the original image has 8 bits per component, add 128. If the original image has 12 bits per component, add 2048. This is the `DetailTexture`.
6. Define and bind the low-resolution texture and the `DetailTexture`. See “Defining and Binding the `DetailTexture`” for instructions.

How DetailTexture is Computed

The GL computes the Level-of-Detail (LOD) at each pixel it textures. LOD is the magnification factor above the base level. LOD n is a 2^n magnification. In the road example, the 512x512 base texture is LOD 0. The DetailTexture combined with the base texture represents LOD 2, which is called the maximum-detail texture.

When a pixel's LOD is between 0 and 2, the GL linearly interpolates between the texture as it looks at LOD 0 and LOD 2. Linearly interpolating between more than 1 LOD can result in aliasing. To minimize aliasing between the known LODs, the GL lets you specify a nonlinear interpolation curve.

Setting the Detail Control Points

Figure 18-15 shows the default linear interpolation and a nonlinear interpolation curve that minimizes aliasing when interpolating between two LODs.

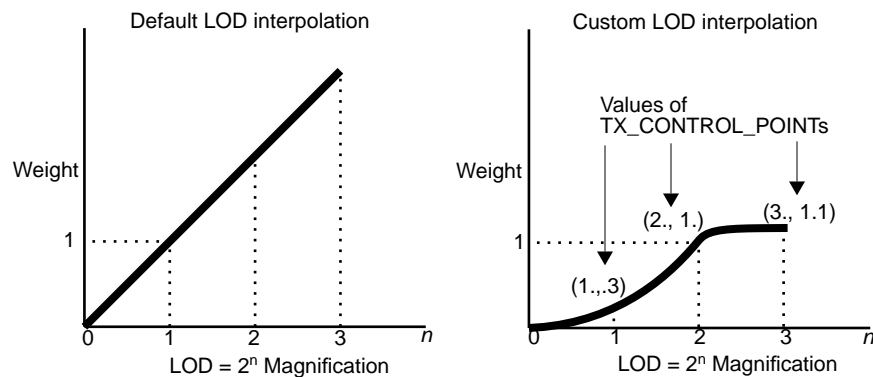


Figure 18-15 LOD Interpolation Curves

The basic strategy is to use very little of the maximum-detail texture until the LOD is within 1 LOD of the maximum-detail texture. More of the information from the maximum-detail texture can be used as the LOD approaches LOD2. At LOD 2, the full amount of detail is used, and the resultant texture exactly matches the high-resolution texture.

Use `TX_CONTROL_POINT` to specify control points for shaping the curve.

The parameters for `TX_CONTROL_POINT` are LOD and weight, where weight is used in the functions listed in Table 18-4 to control how the `DetailTexture` is combined with the base texture.

<code>TX_MAGFILTER</code>	Formula
<code>TX_ADD_DETAIL</code>	$\text{Factor}(n) = \text{weight}(n) * \text{DetailTexture}$
<code>TX_MODULATE_DETAIL</code>	$\text{Factor}(n) = \text{weight}(n) * \text{DetailTexture} * \text{base}$

Table 18-4 Formulas for Computing `DetailTexture` Filters

The following control points can be used to create a nonlinear interpolation, as shown in Figure 18-15, for the road texture example:

```
TX_CONTROL_POINT, 0.0, 0.0,
TX_CONTROL_POINT, 1.0, 0.3,
TX_CONTROL_POINT, 2.0, 1.0,
TX_CONTROL_POINT, 3.0, 1.1,
```

Notice that making the weight at LOD 3 greater than 1.0 extends the extrapolation beyond the maximum-detail texture, which prevents the texture from blurring beyond a 4× magnification.

Defining and Binding the `DetailTexture`

For a texture to be used as a `DetailTexture`, it is bound to the `TX_TEXTURE_DETAIL` target rather than the familiar `TX_TEXTURE_0` target, and used with a texture that has `TX_ADD_DETAIL` or `TX_MODULATE_DETAIL` as a magnification filter.

Use `TX_DETAIL` in the *props* array to define a `DetailTexture`. `TX_DETAIL` is followed by five values, *J*, *K*, *M*, *N*, and *scramble*. *J* and *K* must be equal and *M* and *N* must be equal. Currently, *J* and *K* must both be 4 and *scramble* must be zero.

M and *N* describe the mapping of the `DetailTexture` to the base texture and are given by the following formula:

$$M, N = \frac{256}{2^{8-n}} \tag{EQ 18-1}$$

where *n* is the number of frequency bands, or LODs, in the `DetailTexture`.

In the 2K×2K road texture example, the 256×256 detail texture maps to a 64×64 area of the 512×512 low-resolution texture, so the TX_DETAIL parameters for the detail texture are:

```
TX_DETAIL, 4., 4., 64., 64., 0,
```

The magnification filter for the low-resolution texture is:

```
TX_MAGFILTER, TX_ADD_DETAIL
```

or

```
TX_MAGFILTER, TX_MODULATE_DETAIL
```

When a texture is used as a DetailTexture, the properties MINFILTER, MAGFILTER, MAGFILTER_COLOR, MAGFILTER_ALPHA, TX_WRAP, TX_WRAP_S, TX_WRAP_T, TX_WRAP_R, TX_MIPMAP_FILTER_KERNEL, TX_CONTROL_POINT, TX_CONTROL_CLAMP, and TX_TILE have no effect.

Note: The DetailTexture must have the same number of components and the same number of bits per component as the base texture.

The following code fragment provides another example of how to use a DetailTexture:

To define and bind a DetailTexture, use these properties:

```
TX_DETAIL, 4., 4., 4., 4., 0, TX_NULL
```

To apply a DetailTexture to another texture, use:

```
#define MAX_DETAIL 1.0
```

```
TX_MAGFILTER, TX_MODULATE_DETAIL,  
TX_CONTROL_POINT, 0., 0.0,  
TX_CONTROL_POINT, 0.5, 0.05,  
TX_CONTROL_POINT, 2., 0.4,  
TX_CONTROL_POINT, 5., MAX_DETAIL,  
TX_CONTROL_CLAMP, MAX_DETAIL.
```

```
/* a detail texture must be bound and a base texture must be bound */  
texbind(TX_TEXTURE_DETAIL, detail_texture_id);  
texbind(TX_TEXTURE_0, texture_id);
```

Note: You cannot bind one DetailTexture to another DetailTexture.

18.5 Texture Coordinates

This section describes how to map textures onto object geometry using texture coordinates and how texture coordinates are generated at screen pixels.

To define a texture mapping, you assign texture coordinates to the vertices of a geometric primitive, a process called *parameterization*. You can either assign texture coordinates explicitly with the `t()` subroutines, or let the system automatically generate and assign texture coordinates using the `texgen()` subroutine. You can also use a NURBS texture as described in Chapter 14.

The current texture matrix transforms the texture coordinates. This matrix is set while in `mmode(MTEXTURE)` and is a standard transformation matrix.

The final step generates $(s, t, [r, q])$ at every pixel center inside a geometric primitive by interpolating between the vertex texture coordinates as it fills the geometric primitives during scan conversion.

Note: On RealityEngine, a full 3-D projective transformation is supported.

The IRIS-4D/VGX uses hardware to interpolate texture coordinates linearly. Although hardware interpolation is very fast, it is incorrect for perspective projections. The `scrsubdivide()` subroutine improves interpolation—and consequently image quality—for perspective projections on the VGX.

SkyWriter, VGXT, and RealityEngine systems use an enhanced hardware interpolation that does not require the use of `scrsubdivide()`. IRIS Indigo Entry, XS, XS24, and Elan can perform the perspective correction in software if `getgdesc(GD_TEXTURE_PERSP) = 1`.

18.5.1 Assigning Texture Coordinates Explicitly

Use the `t()` subroutines to specify individual texture coordinates explicitly. The argument you specify for `t()` is a 2-, 3-, or 4-element array whose type can be short, long, float, or double. Like vertex coordinates, texture coordinates can be 2-D, 3-D, or 4-D. Specify the texture coordinates s , t , q , and r in that order for the array. The default for r is 0 and the default for q is 1.

Note: 3-D and 4-D texture coordinates are currently supported only on RealityEngine.

Table 18-5 lists the formats for the `t()` subroutine.

Array Type	2-D	3-D	4-D
Short integer	<code>t2s()</code>	<code>t3s()</code>	<code>t4s()</code>
Long integer	<code>t2i()</code>	<code>t3i()</code>	<code>t4i()</code>
Float	<code>t2f()</code>	<code>t3f()</code>	<code>t4f()</code>
Double	<code>t2d()</code>	<code>t3d()</code>	<code>t4d()</code>

Table 18-5 The `t()` Subroutine

Call the `t()` subroutines within a `bgnpolygon()/endpolygon()` sequence to texture individual vertices, as illustrated below.

```
bgnpolygon();
  t2f (coord1);
  v3f (vertex1);
  t2f (coord2);
  v3f (vertex2);
  t2f (coord3);
  v3f (vertex3);
  t2f (coord4);
  v3f (vertex4);
endpolygon();
```

18.5.2 Generating Texture Coordinates Automatically

The `texgen()` subroutine generates texture coordinates as a function of object geometry. Coordinates are generated on a per-vertex basis and *override* coordinates specified by the `t()` commands. You can independently control the generation of either or both texture coordinates. If you generate only one coordinate, the other is specified by the `t()` subroutines.

`texgen()` can compute the distance of a vertex from a reference plane and calculate texture coordinates proportional to this distance.

The following form of the plane equation is used to define the reference plane:

$$Ax + By + Cz + D = 0 \quad (\text{EQ 18-2})$$

Where the plane normal is the vector

(EQ 18-3)

$$\begin{bmatrix} A \\ B \\ C \end{bmatrix}$$

and the plane constant is D.

For example, the plane $X=Y$ that passes through the origin is $\{1., -1., 0., 0.\}$.

The `TG_LINEAR` mode defines the reference plane in object coordinates so that the parameterization is fixed with respect to object geometry. For example, use `TG_LINEAR` to texture terrain for which sea level is the reference plane. In this case, the altitude of a terrain vertex is its distance from the reference plane. Use `TG_LINEAR` to make the vertex altitude index the texture so that white snow is mapped onto peaks and green grass is mapped onto foothills.

The following code fragment illustrates how to use the `TG_LINEAR` function to generate s coordinates proportional to vertex distance from the object coordinate plane. The first call to `texgen()` defines the generation algorithm for the s coordinate. The second call activates coordinate generation so that the system generates an s coordinate for each vertex.

```
float tgparams[] = {1., -1., 0., 0.};
texgen(TX_S, TG_LINEAR, tgparams);
texgen(TX_S, TG_ON, tgparams);
```

The `TG_CONTOUR` mode defines the specified plane in eye coordinates. The `ModelView` matrix in effect at the time of mode definition transforms the plane equation. Thus, the transformation matrix is not necessarily the same as that applied to vertices. This mode establishes a “field” of texture coordinates that can produce dynamic contour lines on moving objects.

The `TG_SPHEREMAP` mode defines parameters for reflection mapping, by generating texture coordinates based on the vertex and current normal. This causes a reflection of the nearby surrounding environment to map to the surface.

18.5.3 Texture Lookup Tables

This section describes an advanced feature that is available only on RealityEngine systems, so you may want to skip to Section 18.6, "Texture Environments," if you do not have one of these systems.

RealityEngine supports the use of a texture lookup table (TLUT) for translating texture function outputs. Texture function outputs are used by the texture environment to modify the screen pixel color. The texture environment function is defined by `tevdef()` and selected by `tevbind()`.

For textures up to 8-bit per component, 1- or 2-component textures can reference an 8-bit lookup table of 8-bit per component R,G,B,A values to produce full-colored and translucent imagery from intensity textures. This saves memory space and increases the overall texture capacity of the system.

The texture lookup table is defined by `tlutdef()` and selected by `tlutbind()`.

The ANSI C specification for `tlutdef()` is:

```
void tlutdef(long index, long nc, long len,  
            unsigned long *table, long np, float *props)
```

where:

<i>index</i>	is the name of the texture look-up table being defined. Index 0 is reserved as a null definition, and cannot be redefined.
<i>nc</i>	is the number of components per table entry.
<i>len</i>	is the length of table in table entries.
<i>table</i>	is a long-word aligned array of packed <i>nc</i> , 8-bit, component table entries.
<i>np</i>	is the number of symbols and floating point values in <i>props</i> , including the termination symbol <code>TL_NULL</code> . If <i>np</i> is zero, it is ignored. Operation over network connections is more efficient when <i>np</i> is correctly specified, however.
<i>props</i>	is an array of floating point symbols and values that define the texture look-up table. <i>props</i> must contain a sequence of symbols, each followed by the appropriate number of floating point values. The last symbol must be <code>TL_NULL</code> .

The ANSI C specification for `tlutbind()` is:

```
void tlutbind(long target, long index)
```

where:

target is the texture resource to which the texture function definition is to be bound. The only appropriate resource is `TL_TLUT_0`.

index is the name of the texture function that is being bound. Name is the index passed to `texdef2d()` when the texture function is defined.

By default, texture look-up table definition 0 is bound to `TL_TLUT_0`. Texture look-up table use is enabled when a texture function definition other than 0 is bound to `TX_TEXTURE_0`, a texture environment definition other than 0 is bound to `TV_ENV_0`, and a texture look-up table definition other than 0 is bound to `TL_TLUT_0`.

Table 18-6 shows the relationship between the number of components in the texture look-up table, the number of components in the texture, and the resultant action.

TLUT <i>nc</i>	Texture <i>nc</i>	Action
2	1	I looks up I,A
	2	I,A looks up I,A
	3	R,G,B passes through unchanged
	4	R,G,B,A passes through unchanged
3	1	I looks up R,G,B
	2	I,A passes through unchanged
	3	R,G,B looks up R,G,B
	4	R,G,B,A passes through unchanged
4	1	I looks up R,G,B,A
	2	I looks up R,G,B; A looks up A.
	3	R,G,B,B looks up R,G,B,A
	4	R,G,B,A looks up R,G,B,A

Table 18-6 Texture Look-up Table Actions

The following code fragment demonstrates how to use texture lookup tables:

```
maketable4()
{
    int i;
    unsigned long table[256];
    float tlutps[] = {TL_NULL};

    /* inverts colors */
    for (i = 0; i < 256; i++){
        table[i] = ((255-i)<<24) | ((255-i)<<16) | ((255-i)<<8) | ((255-i));
    }
    tlutdef(4,4,256,table,0, tlutps);
    tlutbind(0,4);
}
```

18.5.4 Improving Interpolation Results on VGX Systems

This section describes a technique that applies only to IRIS-4D/VGX systems, so you might want to skip to Section 18.6, “Texture Environments,” if you do not have one of these systems.

On the VGX, texture coordinates are linearly interpolated in screen space by hardware the same way color is interpolated. Although this produces fast rendering, it is mathematically incorrect for perspective projections. For example, you can modify the sample program by replacing the `ortho()` subroutine with `perspective(600, 1., 1., 16.)`, which introduces perspective distortion. Because of incorrect interpolation, textures no longer appear fixed to a surface but shift as the surface moves. This effect is called *swimming*.

Swimming occurs because texture coordinates are interpolated after the perspective division (in screen coordinates) when they should be interpolated in eye coordinates. Because the VGX hardware does not support eye coordinate interpolation, you can use screen subdivision to improve texture coordinate interpolation. Screen subdivision can also improve the accuracy of fog by correctly interpolating w (see Chapter 13).

Note: On systems other than VGX, you *should not* use `scrsubdivide()` because the texture coordinates are already interpolated correctly in eye coordinates.

Use `scrsubdivide` to turn screen subdivision on or off. Use `SS_OFF` to turn off subdivision, which is the default.

Use the `SS_DEPTH` algorithm to subdivide polygons and lines into smaller pieces. Colors, texture coordinates, and the homogeneous coordinate w at newly generated vertices are correctly interpolated in eye coordinates rather than in screen coordinates. Because incorrect interpolation is limited to smaller pieces, error globally decreases and image quality increases. Consequently, you can “tune” image quality by modifying the amount of subdivision.

Note: `scrsubdivide()` is most effective for large, nontessellated polygons and lines. Highly tessellated surfaces (for example, curved surfaces) have, in essence, already been subdivided and thus benefit little from further subdivision.

`SS_DEPTH` subdivision slices screen coordinate polygons and lines by a fixed grid in z . Spacing between z planes is constant throughout the grid and is determined by the three `scrsubdivide()` parameters: maximum screen z , minimum screen size, maximum screen size. The first value in the parameter list specifies the desired distance between subdivision planes in units set by `lsetdepth()`.

If polygon slices generated using this metric span a distance in pixels less than minimum screen size, the distance between subdivision planes is increased until the slices are larger than the minimum screen size. This can occur when a polygon is oriented edge-on, so that it spans little screen distance.

If polygon slices generated using the maximum screen z metric span a distance in pixels greater than maximum screen size, the distance between subdivision planes is decreased until the slices are smaller than the maximum screen size. This parameter is often useful for polygons that face the viewer and suffer from too little subdivision.

In practice, the minimum and maximum screen size parameters are used to keep slices from becoming too small or too big, respectively. However, these parameters can introduce situations in which polygons that share an edge are sliced by differently spaced grids. This generates *T-vertices* that can cause pixel dropout along the shared edges. To avoid T-vertices, you can “turn off” the

screen size parameters by setting them to 0 so that only the maximum screen Z parameter is used. You can turn off any parameter by setting it to 0. For example, a parameter list of {0., 0., 10.} specifies subdivision every 10 pixels.

The following code fragment illustrates how to use screen subdivision:

```
float scrparams[] = {0., 0., 10.};
scrsubdivide(SS_OFF, scrparams);
```

To turn on screen depth subdivision, change the `SS_OFF` mode to `SS_DEPTH`. With the parameter list of {0., 0., 10.}, the quadrilateral is subdivided every 10 pixels and the image quality is improved. You can view the tessellation produced by `scrsubdivide()` by drawing only the polygon outlines, using the `polymode(PYM_LINE)` subroutine (see Chapter 2).

18.6 Texture Environments

A texture environment specifies how texture values modify the color and opacity of an incoming shaded pixel. Use the `tevdef()` subroutine to define a texture environment and the `tevbind()` subroutine to enable the texturing environment. As with `texbind()`, there can be only one texture environment bound (active) at a time.

There are three texture environment types:

- | | |
|--------------------------|---|
| <code>TV_MODULATE</code> | Modulates the polygon surface with the texture. This is the default environment and is valid for 1-, 2-, 3-, and 4-component textures. |
| <code>TV_BLEND</code> | Interpolates between the polygon color and a constant color based on the texel intensity. This environment is valid for 1- and 2-component textures only. |
| <code>TV_DECAL</code> | Applies the texture on top of the polygon color wherever texture alpha is nonzero. |

The ANSI C specification for `tevdef()` is:

```
tevdef(long index, long np, float props[])
```

where:

index is the unique index(name) for the texture environment.

np is the number of elements in the *props* array.

props is a array of floating point constants that defines how the texture is combined with incoming pixels to color screen pixels.

The texture environment function takes a shaded, incoming pixel color ($R_{in}, G_{in}, B_{in}, A_{in}$) and computed texture values ($R_{tex}, G_{tex}, B_{tex}, A_{tex}$) as input, and outputs a new color ($R_{out}, G_{out}, B_{out}, A_{out}$). The equations for each environment are listed in the tables that follow the environment description.

TV_MODULATE multiplies the incoming color components by texture values, according to the equations listed in Table 18-7.

1-component	2-component	3-component	4-component
$R_{out} = R_{in} \cdot I_{tex}$	$R_{out} = R_{in} \cdot I_{tex}$	$R_{out} = R_{in} \cdot R_{tex}$	$R_{out} = R_{in} \cdot R_{tex}$
$G_{out} = G_{in} \cdot I_{tex}$	$G_{out} = G_{in} \cdot I_{tex}$	$G_{out} = G_{in} \cdot G_{tex}$	$G_{out} = G_{in} \cdot G_{tex}$
$B_{out} = B_{in} \cdot I_{tex}$	$B_{out} = B_{in} \cdot I_{tex}$	$B_{out} = B_{in} \cdot B_{tex}$	$B_{out} = B_{in} \cdot B_{tex}$
$A_{out} = A_{in}$	$A_{out} = A_{in} \cdot A_{tex}$	$A_{out} = A_{in}$	$A_{out} = A_{in} \cdot A_{tex}$

Table 18-7 TV_MODULATE Equations

TV_BLEND blends the incoming color and the active texture environment color, which is a single RGBA constant ($R_{const}, G_{const}, B_{const}, A_{const}$) according to the equations used for TV_BLEND in Table 18-8. The texture environment color is specified with the TV_COLOR parameter.

Output Color	1-component	2-component
Red	$R_{out} = R_{in} \cdot (1 - I_{tex}) + R_{const} \cdot I_{tex}$	$R_{out} = R_{in} \cdot (1 - I_{tex}) + R_{const} \cdot I_{tex}$
Green	$G_{out} = G_{in} \cdot (1 - I_{tex}) + G_{const} \cdot I_{tex}$	$G_{out} = G_{in} \cdot (1 - I_{tex}) + G_{const} \cdot I_{tex}$
Blue	$B_{out} = B_{in} \cdot (1 - I_{tex}) + B_{const} \cdot I_{tex}$	$B_{out} = B_{in} \cdot (1 - I_{tex}) + B_{const} \cdot I_{tex}$
Alpha	$A_{out} = A_{in}$	$A_{out} = A_{in} \cdot A_{tex}$

Table 18-8 TV_BLEND Equations

TV_COLOR	specifies the constant color used by the TV_BLEND environment. Four floating point values, in the range 0.0 through 1.0, must follow this symbol. These values specify R_{con} , G_{con} , B_{con} , and A_{con} . By default, all are set to 1.0.
TV_DECAL	uses texture alpha (referred to as A_{tex} in the equations below) is used to blend the incoming color and the texture color, according to the blend equations in Table 18-9

Output Color	3-component	4-component
Red	$R_{out} = R_{tex}$	$R_{out} = R_{in} \cdot (1 - A_{tex}) + R_{tex} \cdot A_{tex}$
Green	$G_{out} = G_{tex}$	$G_{out} = G_{in} \cdot (1 - A_{tex}) + G_{tex} \cdot A_{tex}$
Blue	$B_{out} = B_{tex}$	$B_{out} = B_{in} \cdot (1 - A_{tex}) + B_{tex} \cdot A_{tex}$
Alpha	$A_{out} = A_{in}$	$A_{out} = A_{in}$

Table 18-9 TV_DECAL Equations

TV_COMPONENT_SELECT allows the use of one or two components from a texture with more components. Some GL implementations may allow 4 component textures with a very small component size, such as 4 bits, which is smaller than the smallest addressable datum. Therefore, a 4 component texture with 4 bits per component may be used as four separate 1-component textures, or two 2-component textures, and so on.

The token is followed by one choice from the following:

TV_I_GETS_R	uses the red component of a 4-component texture as a 1-component texture.
TV_I_GETS_G	uses the green component of a 4-component texture as a 1-component texture.
TV_I_GETS_B	uses the blue component of a 4-component texture as a 1-component texture.
TV_I_GETS_A	uses the alpha component of a 4- or 2-component texture.

18.7 Texture Programming Hints

After you understand the basics of the IRIS GL texture routines, the following hints can be useful in getting the optimal performance from your system.

Most of these hints apply to RealityEngine, but because of the texturing capabilities of RealityEngine, some do not apply and are so noted. RealityEngine systems feature dedicated texture memory, rather than using framebuffer memory for textures. This provides the ability to display complex, texture-mapped scenes at fast frame rates, thereby improving image quality without sacrificing performance.

RealityEngine provides 4Mbytes of standard, on-line texture memory, stored as two banks of 2Mbyte memory areas. Different texture types and modes can be mixed together within the memory storage space. Built-in texture storage algorithms store textures sequentially in memory for maximum efficiency. The minimum texture size on RealityEngine is 2×2, and the maximum size is 1024×1024. The system can store two R,G,B,A full-color 1024×1024 textures with 16-bit texels.

Overall Hints

- Turn off texturing when you are not drawing textured geometry.

Remember to turn off texturing when drawing nontextured geometry. Not supplying texture coordinates does *not* disable texturing. Texturing is disabled only with one of the following subroutine calls: `texbind(TX_TEXTURE_0, 0)` or `tevbind(TV_ENV0, 0)`.

- Texturing works only in RGB mode.

The behavior of texturing is not defined in color index mode.

- Most texture calls are illegal between `bgn/end` sequences.

With the exception of the `t()` commands, the texture subroutines described in this chapter cannot be called inside of `bgn/end` sequences such as `bgnpolygon()/endpolygon()`.

Hints for Using `texdef2d`

- Use images whose dimensions are powers of two whenever possible.

Internally, the GL works only with images whose dimensions are powers of 2. `texdef2d` automatically resizes images as necessary. To avoid resizing, pass `texdef2d()` images that have widths and heights that are powers of 2.

- Use as few components as necessary.

The more components a texture has, the longer it takes to map the texture onto a polygon. For optimal speed, use as few components as possible. This applies to RealityEngine unless you specify an internal format. If you are not taking advantage of a texture's alpha, define the texture as a 1- or 3-component texture. If you do not need a full-color texture, define the texture with one or two components.

- Use the simplest filter you need.

The per-pixel speed of the texture filter functions is related to the number of interpolations the filter has to perform. The filters in order from fastest to slowest are `TX_POINT`, `TX_MIPMAP_POINT`, `TX_MIPMAP_LINEAR`, `TX_BILINEAR`, `TX_MIPMAP_BILINEAR`, `TX_MIPMAP_TRILINEAR` and `TX_TRILINEAR`, `TX_BICUBIC` and `TX_MIPMAP_QUADLINEAR`.

There is some overhead per polygon for using MIPmap filters. If a scene has a large number of textured polygons, or if the polygons are subdivided finely, performance is improved if MIPmap filters are not used.

- Keep the texture size below the recommended maximum.

Textures that exceed the maximum dimensions of the graphics hardware are resized to the maximum dimensions. The maximum dimensions for textures using MIPmapping are half of those that do not. The effect is that large textures using MIPmapping are more blurry than those that do not.

- Be aware that `texdef2d()` copies the texture image to memory.

The image passed to `texdef2d()` is copied. All other data associated with the texture, such as its MIPmap, are saved with it in the user's memory space until the texture is redefined or the program exits.

Hints for Using `texbind`

- Bind textures as infrequently as possible.

`texbind` can be a time-consuming operation, especially if the texture is not resident in the graphics hardware. To achieve maximum performance, draw all of the polygons that use the same texture together.

- Texture caching.

Hardware texture memory is a finite resource managed by the IRIX kernel. The kernel guarantees that the currently bound texture of a program resides in this memory, whenever the program owns the graphics pipe. Beyond that, the kernel keeps as many additional textures as possible in the hardware texture memory. When a texture is bound, if it is not resident in the texture memory and there is not enough room remaining for this texture, one or more of the resident textures are swapped out. To minimize the frequency of this swapping, use smaller textures or try to switch them less often.

Hints for Using `scrsubdivide`

- Use `scrsubdivide()` only on VGX systems.

- Turn on `scrsubdivide()` only when you need it.

Because `scrsubdivide()` generates many polygons from each incoming polygon, it is wise to turn off this feature when it is not needed, such as when you are drawing non-texture-mapped polygons or highly tessellated texture-mapped polygons.

- Use only as much subdivision as you need.

Choose the `scrsubdivide()` parameters carefully. For maximum performance, use only as much subdivision as is necessary. Textures without high frequencies need less subdivision than those with high frequencies.

Hints for Using `alpha`

- Use `afunction()`, and/or `msalpha()` on RealityEngine for fast drawing of objects with texture alpha.

When the alpha component of a texture is used to approximate a geometry (such as when a texture is used to describe a tree), the polygons must be blended into the scene in sorted order to properly realize the coverage defined by the alpha component. This sorting and blending requirement can be removed by using `afunction()`. `afunction(0, AF_NOTEQUAL)` specifies that only pixels with nonzero alpha be drawn. See the `afunction()` man page for more details. On VGXT, `afunction(128, AF_GREATER)` works well.

- Alphaless systems.

Systems without alpha memory also lack storage for a fourth texture component. On such systems, the alpha component of 4-component textures always appears to be 255. 1- and 3-component textures behave the same on systems with or without alpha.

18.8 Sample Texture Programs

This sample program, *brick.c*, creates a brick texture and lets you toggle `scrsubdivide()` with the mouse, to view texture “swimming.”

```
#include <stdio.h>
#include <gl/gl.h>
#include <gl/device.h>

float texprops[] = {TX_MINFILTER, TX_POINT, TX_MAGFILTER, TX_POINT,
                   TX_WRAP, TX_REPEAT, TX_NULL};

/* Texture color is brick-red */
float tevprops[] = {TV_COLOR, .75, .13, .06, 1., TV_BLEND, TV_NULL};

/* Subdivision parameters */
float scrparams[] = {0., 0., 10.};

unsigned long bricks[] = /*Define texture image */
    {0x00ffffff, 0xffffffff,
     0x00ffffff, 0xffffffff,
     0x00ffffff, 0xffffffff,
     0x00000000, 0x00000000,
     0xffffffff, 0x00ffffff,
     0xffffffff, 0x00ffffff,
     0xffffffff, 0x00ffffff,
     0x00000000, 0x00000000};
```

```

/* Define texture and vertex coordinates */
float t0[2] = {0., 0.}, v0[3] = {-2., -4.,0.};
float t1[2] = {16., 0.}, v1[3] = {2., -4.,0.};
float t2[2] = {16., 32.}, v2[3] = {2., 4.,0.};
float t3[2] = {0., 32.}, v3[3] = {-2., 4.,0.};

main()
{
    short val;
    int dev, texflag;

    if (getgdesc(GD_TEXTURE) == 0) {
        fprintf(stderr, "texture mapping not available on this machine\n");
        return 1;
    }
    keepaspect(1, 1);
    winopen("brick");
    subpixel(TRUE);
    RGBmode();
    doublebuffer();
    gconfig();
    qdevice(ESCKEY);
    qdevice(LEFTMOUSE);
    qenter (LEFTMOUSE, 0);
    mmode(MVIEWING);
    perspective(600, 1., 1., 10.);
    texdef2d(1, 1, 8, 8, bricks, 0, texprops);
    tevdef(1, 0, tevprops);
    texbind(TX_TEXTURE_0, 1);
    tevbind(TV_ENV0, 1);
    texflag = getgdesc(GD_TEXTURE_PERSP);
    translate(0., 0., -6.);          /* Move poly away from viewer */

    while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
        while (TRUE){
            while(qtest()){
                dev = qread(&val);
                switch(dev){
                    case ESCKEY: exit(0);
                                break;
                    case REDRAW: reshapeviewport();
                                break;
                }
            }
        }
    }
}

```

```

/* Screen subdivision - use it only if you have a VGX.
Push the leftmouse button to see "swimming" on VGX's */
case LEFTMOUSE:
    if (val){
        switch(texflag){
            case 0: scrsubdivide(SS_OFF, scrparams);
                    break;
            case 1: printf("Your machine corrects in hardware\n");
                    break;
        }
    }
    else
        switch(texflag){
            case 0: scrsubdivide(SS_DEPTH, scrparams);
                    break;
            case 1: break;
        }
        break;
    } /* end main switch */
} /* end qtest */
cpack(0x0);
clear();
pushmatrix();
rotate(getvaluator(MOUSEX)*5, 'y');
rotate(getvaluator(MOUSEY)*5, 'z');
cpack(0xffccccc);
bgnpolygon();
    t2f(t0); v3f(v0);
    t2f(t1); v3f(v1);
    t2f(t2); v3f(v2);
    t2f(t3); v3f(v3);
endpolygon();
popmatrix();
swapbuffers();
}
texbind(TX_TEXTURE_0, 0); /* Turn off texturing */
}

```

This sample program, *heat.c*, illustrates texture mapping in color map mode.

```
#include <stdio.h>
#include <gl/gl.h>
#include <gl/device.h>

/* Texture environment */
float tevprops[] = {TV_MODULATE, TV_NULL};

/* RGBA texture map representing temperature as color and
 * opacity */
float texheat[] = {TX_WRAP, TX_CLAMP, TX_NULL};
/* Black->blue->cyan->green->yellow->red->white */
unsigned long heat[] = /* Translucent -> Opaque */
    {0x00000000, 0x55ff0000, 0x77ffff00, 0x9900ff00,
     0xbb00ffff, 0xdd0000ff, 0xffffffff};

/* Point sampled 1 component checkerboard texture */
float texbgd[] = {TX_MAGFILTER, TX_POINT, TX_NULL};

unsigned long check[] =
    {0xff800000, /* Notice row byte padding */
     0x80ff0000};

/* Subdivision parameters */
float scparams[] = {0., 0., 10};

/* Define texture and vertex coordinates */
float t0[] = {0., 0.}, v0[] = {-2., -4., 0.};
float t1[] = {.4, 0.}, v1[] = { 2., -4., 0.};
float t2[] = {1., 0.}, v2[] = { 2., 4., 0.};
float t3[] = {.7, 0.}, v3[] = {-2., 4., 0.};
```



```

main()
{
    long    device;
    short   data, sub = 0;

    if (getgdesc(GD_TEXTURE) == 0){
        fprintf(stderr,
            "Texture mapping not available on this machine\n");
        return 1;
    }
    keepaspect(1,1);
    winopen("heat");
    RGBmode();
    doublebuffer();
    gconfig();
    subpixel(TRUE);
    lsetdepth(0x0, 0x7fffffff);

    blendfunction(BF_SA, BF_MSA); /* Enable blending */

    mmode(MVIEWING);
    perspective(600, 1, 1., 16.);

    /* Define checkerboard */
    texdef2d(1, 1, 2, 2, check, 0, texbgd);
    /* Define heat */
    texdef2d(2, 4, 7, 1, heat, 0, texheat);
    tevdef(1, 0, tevprops);
    tevbind(TV_ENV0, 1);

    translate(0., 0., -6.);
    qdevice(ESCKEY);

    /* Determine if machine does perspective correction */
    if (getgdesc(GD_TEXTURE_PERSP) != 1) sub = 1;
}

```

```

while(TRUE) {
    if(qtest()){
        device = qread(&data);
        switch(device){
            case ESCKEY: texbind(TX_TEXTURE_0, 0); /* Turn off texturing */
                        exit(0);
                        break;
            case REDRAW: reshapeviewport();
                        break;
        }
    }
    cpack(0x0);
    clear();

    /* Subdivision off */
    if (sub) scrsubdivide(SS_OFF, scrparams);
    texbind(TX_TEXTURE_0, 1); /* Bind checkerboard */
    cpack(0xff102040); /* Background rectangle color */

    bgnpolygon(); /* Draw textured rectangle */
    t2f(v0); v3f(v0); /* Notice vertex */
    t2f(v1); v3f(v1); /* coordinates are used */
    t2f(v2); v3f(v2); /* as texture coordinates */
    t2f(v3); v3f(v3);
    endpolygon();

    pushmatrix();
    rotate(getvaluator(MOUSEX)*5, 'y');
    rotate(getvaluator(MOUSEY)*5, 'x');

    /* Screen subdivision - use it only if you have a VGX */
    if (sub) scrsubdivide(SS_DEPTH, scrparams);
    texbind(TX_TEXTURE_0, 2); /* Bind heat */
    cpack(0xffffffff); /* Heated rectangle base color */
    bgnpolygon(); /* Draw textured rectangle */
    t2f(t0); v3f(v0);
    t2f(t1); v3f(v1);
    t2f(t2); v3f(v2);
    t2f(t3); v3f(v3);
    endpolygon();

    popmatrix();
    swapbuffers();
}
}

```

Using the GL in a Networked Environment

Network transparency is a built-in feature of the GL that allows a process on one IRIS workstation to display graphics either locally or over the network on a remote IRIS workstation.

19.1 Introduction

The network-transparent feature of the GL lets systems share the work load for graphics applications and lets servers without graphics capabilities use graphical tools.

For example, consider running a flight simulation to test a new aircraft design. You want to run a complex mechanical analysis with a simultaneous real-time animation. The mechanical analysis requires a “number-crunching” system and the animation requires a fast graphics display system. The two systems can share the work load, each doing the task for which it is best suited, in a client-server relationship, resulting in a more balanced work load and better overall performance.

The client/server model of the network-transparent GL allows remote display of graphics output. In the above example, a 4Server, acting as the client, performs the calculations for the mechanical analysis and sends the graphics calls over the network to an IRIS-4D workstation, acting as the graphics server, to display the flight animation.

19.1.1 Protocol

Network transparency is based on the Distributed GL (DGL) protocol that is built into the shared GL. The DGL protocol has two parts:

- a call mechanism built into the shared GL
- a graphics server to service requests made by DGL clients

In this chapter, the client application, which is linked with the shared GL, is called the *DGL client* and the graphics server is called the *DGL server*. In the DGL client, the DGL protocol sends tokens in a byte stream to the graphics server over the Ethernet® or other communication medium. The graphics server decodes the byte stream and calls the GL subroutines to display the graphics.

There is a separate product for running GL applications on non-IRIS hosts; see the documentation that comes with that option for more information.

19.1.2 Writing GL Programs to Use Network Transparent Features

Existing GL programs do not contain any calls that specifically invoke the DGL server. However, these programs can still be run remotely without modifying the source code, simply by relinking them with the shared GL (`-lg1_s`) and by linking with the Sun library (`-lsun`) if the Network Information Service (NIS) is desired.

Writing a network-transparent GL program is no different than writing a standalone GL program, except for optimizing performance.

Graphics calls are buffered from the client to the server, so you must flush the buffer periodically. The subroutine `gflush()` flushes the client buffer so GL calls can be received by the server.

gflush

The DGL client buffers calls to GL subroutines for efficient block transfer to the graphics server. The subroutine `gflush()` explicitly flushes the communication buffers and delivers all the untransmitted graphics data that is in the buffer to the graphics server.

GL subroutines that return data implicitly flush the communication buffers. In most programs, the implicit flushing that is performed by subroutines that return data is usually sufficient.

Note: All programs that are run over the network must call `gflush()` if the last command is a drawing command. No drawing is guaranteed to happen until `gflush()` is called.

The following situation illustrates a typical use of `gflush()`:

A program calls some Graphics Library subroutines that are buffered and not flushed. The program then either computes or blocks for a while, waiting for non-graphic I/O. `gflush()` must be called if the results of the buffered GL subroutines are to be seen on the host display before and during the pause.

Another reason for using `gflush()` is to reduce display jerkiness. If the client is computing data and then sending the data to the graphics server without implicit or explicit flushes, the data will arrive at the graphics server in large batches. The server may process this data very quickly and then wait for the next large batch of data. The rapid processing of GL subroutines followed by a pause results in an undesirable “jerky” appearance. In these cases it is probably best to call `gflush()` periodically. For example, a logical place to call `gflush()` is after every `swapbuffers()` call.

Note: Performing too many flushes can adversely effect performance.

finish

`finish()` is useful when there are large network and pipeline delays. `finish()` blocks the client process until all previous subroutines execute. First, the communication buffers on the client machine are flushed. On the graphics server, all unsent subroutines are forced down the Geometry Pipeline to the bitplanes, then a final token is sent and the client process blocks until the token goes through the pipeline and an acknowledgment is sent to the graphics server and forwarded to the client process.

The following example illustrates a typical use of `finish()`:

A client calls GL subroutines to display an image. The subroutines all fit into the server’s network buffers and the image takes 30 seconds to render. The client wants to wait until the image is completely displayed on the server’s monitor before a message can be displayed on the client’s terminal. `gflush()`

flushes the buffers, but does not wait for the server to process the buffers. `finish()` flushes the buffers and waits not only for the server to process all the graphics subroutines, but for the Geometry Pipeline to finish as well.

19.1.3 Establishing a Connection

To establish a connection, the client must have permission to connect to the graphics server. Permission is verified as it is for X clients. See the *xhost* man page for more information about client authentication procedures.

A server connection is established according to these rules:

1. If any of the following environment variables is defined, the server name is the value of the defined variable highest in the following list:
 1. DISPLAY
 2. DGLSERVER
 3. REMOTEHOST
2. If none of these environment variables are defined, then the server name is set to the client's hostname.

Note: The environment variables *DGLTYPE* and *DGLTsocket* are used for Silicon Graphics internal debugging purposes.

19.1.4 Using rlogin

If you use *rlogin* to log in remotely to an IRIS workstation, *REMOTEHOST* is defined. If *DGLSERVER* is undefined, the DGL protocol by default establishes a connection back to the last remote system where you ran *rlogin*. For example, if you *rlogin* from system A to system B and then *rlogin* from system B to system C, *REMOTEHOST* is set to B on system C. In this example the default graphics connection is B.

19.2 Limitations and Incompatibilities

The network-transparent GL has a few limitations and incompatibilities with the previous releases of the GL that was used strictly for local imaging. These limitations may prevent a GL application from executing properly only when remote connections are used.

19.2.1 The `callfunc` Routine

The `callfunc()` subroutine does not function in a GL program that is run remotely. Any references to `callfunc()` will result in a run-time error when executing the program.

19.2.2 Pop-up Menu Functions

A maximum of 16 unique callback functions are supported. Freeing pop-up menus does not free up callback functions. If you use too many callback functions, you get the client error:

```
dgl error (pup): too many callbacks
```

19.2.3 Interrupts and Jumps

You cannot interrupt the execution of a remotely called GL subroutine or pop-up menu callback function without returning back to that subroutine before calling another subroutine. This illegal condition typically results when you set an alarm or timer interrupt to go off and then block the program with a `qread()` call. If the signal handler does not return to the `qread()`, unpredictable results are likely (for example, it does a *longjmp*(3C) to some non-local location).

19.3 Using Multiple Server Connections

Connections to multiple graphics servers from one GL client program are supported and there are mechanisms for creating, multiplexing, and destroying server connections. You can use GL or mixed-model (X Window System and GL) subroutines for managing multiple server connections. Server processes normally reside on different server machines, but they can also reside on the same machine.

There are advantages to using multiple graphics servers, for example, some applications may require multiple windows, each with very high resolution graphics. Multiple windows on the same server must share one screen's resolution; however, with the network transparent feature of the GL, an application can control multiple servers, each of which can devote its full screen resolution to its windows.

Another possible application for multiple servers is improving performance when displaying multiple views of complex objects. If multiple views are displayed on multiple servers, performance is linearly increased by the number of servers. For example, an application can create a display list for a car on each of the servers that includes material and lighting parameters. Each server is given a different set of viewing parameters and then used to display the object.

A slight variation of the previous example is to have each server display a different representation of the object. For example, one server displays a depth-cued wireframe mesh of the car, another server displays a flat shaded polygonal representation of the car, and a third server displays a smooth shaded lighted surface representation of the car. If the display list for each of these representations is very large, multiple servers can eliminate or reduce paging, because each server needs only the display list for its representation.

19.3.1 Establishing and Closing Multiple Network Connections

The subroutines `dglopen()` and `dg1close()` allow a GL program to open and close graphics connections to server machines. You don't have to use these subroutines if your application is running on a single server because there is a default connection procedure, but you must use them if you are connecting to multiple servers.

Using `dglopen` to Open a Connection

`dglopen()` opens a connection to a graphics server, and makes it the current connection. After a connection is established, all graphics preferences, input, and output are directed to that connection.

Communication remains enabled over the connection either until the connection is closed, or until a different connection is opened. A remote connection is closed by calling `dglclose()` with the *server identifier* returned by `dglopen()`. A different connection is selected by calling a subroutine that takes a graphics window identifier as an input parameter. The server connection associated with that graphics window identifier becomes the current connection.

To establish a connection, the client host must have permission to connect to the graphics server. Permission is verified as it is for X clients (see the *xhost* man page for more information about client authentication procedures).

To open a connection, you call `dglopen()` with a pointer to the server name (*svname*) and the type of connection you want.

Specify the server name as follows:

```
[ [username]password@]hostname[:server[.screen]]
```

The *username* and *password* parameters are ignored; they are included for compatibility only. The *hostname* must be an Internet host name recognized by *gethostbyname*. If the connection succeeds, `dglopen()` returns the *server identifier*, a non-negative integer. Otherwise, `dglopen()` indicates a failure by returning a negative integer, the absolute value of which indicates the reason for failure.

Two types of connections are supported:

- `DGLLOCAL` is a direct connection to the local graphics hardware. This type of connection is not supported on client systems without IRIS graphics hardware.
- `DGLTsocket` is a TCP/IP socket connection to a remote host.

Because you can mimic the behavior of a remote connection by using a `DGLTsocket` connection on a single machine, you can use the `DGLTsocket` connection during the development process to debug a remote application without connecting to another machine.

The following sequence of events occurs when a DGLTsocket connection is attempted:

1. The service *sgi-dgl* is looked up in */etc/services* to get a port number. If the service is not found, then an error occurs.
2. The server's name is looked up in */etc/hosts* or by the Network Information Service (NIS) to get an Internet address. If the host is not found, then an error occurs.
3. An Internet stream socket is created and some of its options are set.
4. A connection to the server machine is attempted with a small time-out. If the connection is refused, the timeout is doubled and the connection retried. If after several tries, the connection is still refused, an error occurs.
5. A successful connection is made and the server's Internet daemon invokes a copy of the graphics server. The graphics server process inherits the socket for communicating with the client program.
6. The graphics server uses the X authentication model to verify the login. Authentication is accomplished by the same mechanism as for X clients. See *xhost(1)* for more details.
7. The server process's group and user ID are changed according to the entry in */etc/passwd*.

Using *dglclose* to Close a Connection

To destroy a graphics server process and its connection, call *dglclose()* with the *server identifier* returned by *dglopen()*. This terminates the graphics server process, freeing system resources, for example, open windows, that had been allocated and closes the graphics connection, freeing associated system resources on the client machine. Calling *dglclose()* with a negative *server identifier* closes all graphics server connections.

After *dglclose()*, there is no current connection. In order to resume rendering, you have to select another valid connection by calling a routine that takes a graphics window id as a parameter (such as *winset*) or you have to open another connection with *dglopen()*. Although it is not necessary, it is recommended that *dglclose(-1)* be called before exiting a GL application. This ensures that the graphics server processes exit cleanly.

19.3.2 Graphics Input

Each graphics server has its own keyboard, mouse, and optional dial and button box. The graphics input subroutines `qtest()`, `qread()`, `qdevice()`, `getvaluator()`, `setvaluator()`, and `noise()` execute on the current graphics server. The client program can therefore solicit input from multiple keyboards and mice. For most programs, it will make sense to get input from only one graphics server. In all cases, the programmer must make sure that the connection to the current graphics server is set correctly when graphics input is solicited.

19.3.3 Local Graphics Data

Each server process runs a separate copy of the GL and has its own local set of graphics data. For example, linestyles, patterns, fonts, materials, lights, and display list objects are local to each graphics server. When graphics data is defined, it is defined only on the current graphics server; other servers do not define it. You must be careful to reference local graphics data only on the server where it is defined. If a display list or font is used on multiple servers, it must be defined on each server.

19.3.4 Sample Program - Multiple Connections on a Local Host

This sample program illustrates how to establish multiple connections on a local host to solicit multiple graphics input.

```
#include <stdio.h>
#include <gl/gl.h>
#include <gl/device.h>
#include <sys/types.h>
#include <sys/time.h>

static void DoLoop();

long main(int argc, char *argv[])
{
    int i;
    long wid1, wid2;
    fd_set readfds;
    long gl_fd1, gl_fd2;
    int nfound;
```

```

dglopen ("", DGLTsocket); /* force socket connection to local host */
wid1 = winopen("win 1");
qdevice(INPUTCHANGE);
qdevice(LEFTMOUSE);
qdevice(ESCKEY);
qdevice(REDRAW);

RGBmode();
gconfig();
cpack(0xff00ff);
clear();
finish();

dglopen ("", DGLTsocket); /* force socket connection to local host */
wid2 = winopen("win 2");
qdevice(INPUTCHANGE);
qdevice(LEFTMOUSE);
qdevice(ESCKEY);
qdevice(REDRAW);

RGBmode();
gconfig();
cpack(0x00ffff);
clear();
finish();

FD_ZERO(&readfds);

winset (wid1);
if ((gl_fd1 = qgetfd()) < 0) {
    printf("bad file descriptor %d\n", gl_fd1);
    exit(-1);
}

winset (wid2);
if ((gl_fd2 = qgetfd()) < 0) {
    printf("bad file descriptor %d\n", gl_fd2);
    exit(-1);
}

while(1) {
    FD_SET(gl_fd2, &readfds);
    FD_SET(gl_fd1, &readfds);
    nfound = select (getdtablesize(), &readfds, 0, 0, 0);
    printf("select nfound = %d\n", nfound);
}

```

```

        if (FD_ISSET(gl_fd1, &readfds)) {
            winset(wid1);
            DoLoop();
        }

        if (FD_ISSET(gl_fd2, &readfds)) {
            winset(wid2);
            DoLoop();
        }
    }
}

static void DoLoop()
{
    long dev;
    short val;

    while (qtest()) {
        dev = qread(&val);
        switch(dev) {
            case INPUTCHANGE:
                printf("INPUTCHANGE; wid = %d\n", val);
                break;
            case LEFTMOUSE:
                printf("LEFTMOUSE; val = %d\n", val);
                break;

            case REDRAW:
                printf("REDRAW; wid = %d\n", val);
                winset(val);
                clear();
                finish();
                break;

            case ESCKEY:
                gexit();
                exit(-1);
        }
    }
}

```

19.4 Configuration of the Network Transparent Interface

The DGL protocol software consists of two parts: a client library and a graphics server daemon. The client library is built into the shared GL (*/usr/lib/libgl_s.a*) and the graphics server daemon is */usr/etc/dgld*. The DGL protocol gets an Internet port number from */etc/services*, which has an entry for *sgi-dgl* (see *services(4)*).

19.4.1 inetd

The graphics server daemon for TCP socket connections is automatically started by *inetd(1M)*. *inetd* reads its configuration file to determine which server programs correspond to which sockets. The standard configuration file, */usr/etc/inetd.conf*, has an entry for *sgi-dgl*. When a request for a connection is made:

1. The service *sgi-dgl* is looked up in */etc/services* to get a port number. If the service is not found, then an error occurs.
2. The server's name is looked up in */etc/hosts* or by the Network Information Service (NIS) to get an Internet address. If the host is not found, then an error occurs.
3. An Internet stream socket is created and some of its options are set.
4. A connection to the server machine is attempted with a small timeout allowance. If the connection is refused, the timeout is doubled and the connection retried. If after several tries, the connection is still refused, an error occurs.
5. A successful connection is made and the server's Internet daemon invokes a copy of the DGL graphics server. The graphics server process inherits the socket for communicating with the DGL client program.
6. The graphics server uses the X authentication model to verify the login. Authentication is accomplished by the same mechanism as for X clients (see *xhost(1)*).
7. The server process's group and user ID are changed according to the entry in */etc/passwd*.

19.4.2 **dgl**

The *dgl* daemon is the server for remote graphics clients. The server provides both a subprocess facility and a networked graphics facility. *dgl* is started by *inetd* when a remote request is received.

Local connections are not controlled by *dgl*; instead, a client program running on an IRIS host calls GL subroutines directly on the host machine. No authentication is performed for local connections.

TCP socket connections are serviced by the Internet server daemon *inetd*. *inetd* listens for connections on the port indicated in the *sgi-dgl* service specification. When a connection is found, *inetd* starts *dgl* as specified by the file */usr/etc/inetd.conf* and gives it the socket.

19.5 **Error Messages**

Error messages are output to a message file. The message file defaults to *stderr*. Error messages have the following format:

```
pgm-name error (routine-name): error-text
```

pgm-name is either *dgl* for client errors or *dgl* for server errors.

routine-name is the name of the system service or internal routine that failed or detected the error.

error-text is an explanation of the error.

19.5.1 Connection Errors

Table 19-1 lists the internally generated error values (defined in `<errno.h>`) that are reported when a connection fails.

Error Value	Explanation
ENODEV	type is not a valid connection type
EACCESS	login incorrect or permission denied
EMFILE	too many graphics connections are currently open
ENOPROTOOPT	DGL service not found in <code>/etc/services</code>
EPROTONOSUPPORT	DGL version mismatch
ERANGE	invalid or unrecognizable number representation
ESRCH	window manager is not running on the graphics server

Table 19-1 Error Values

19.5.2 Client Errors

Client error messages are output to `stderr`. For example, if NIS is not enabled and `/etc/hosts` does not include an entry for the server host `foobar`, the following error message is output when a connection to is requested:

```
dgl error (gethostbyname): can't get name for foobar
```

If the client detects a condition that is fatal, it exits with an `errno` value that best indicates the condition. If a system call or service returns an error number (`errno` or `h_errno`), this number is used as the exit number.

Table 19-2 lists all exit values that are internally generated (not the result of a failed system call or service).

Exit Value	Explanation
ENOMEM	out of memory
EIO	read or write error

Table 19-2 DGL Client Exit Values

The EIO value, accompanied by the message

```
dgl error (comm): read returned 0
```

usually means that communication with the server has been interrupted or was not successfully established. The configuration of the server machine should be checked (see Section 19.4).

19.5.3 Server Errors

Server error messages are output to *stderr* by default. For example, if */etc/hosts* does not include an entry for the client host, the following error messages are be output:

```
dgld error (gethostbyaddr): can't get name for 59000002  
dgld error (comm_init): fatal error 1
```

The standard *inetd.conf* file runs the graphics server with the **I** and **M** options. The **I** option informs the graphics server that it was invoked from *inetd* and enables output of all error messages to the system log file maintained by *syslogd*(1M). The **M** option disables all message output to *stderr*.

If the DGL server is not working properly, check the system log file for error messages. Each entry in the *SYSLOG* file includes the date and time, identifies the program as *dgld*, and includes the process identification number (PID) for the server process. The rest of the error message is the text of the error message.

19.5.4 Exit Status

When the *dgl* graphics server exits, the exit status indicates the reason for the exit. A normal exit has an exit status of zero. A normal exit occurs when either the client calls `dglclose()` or when zero bytes are read from the graphics connection. The latter case can occur when the client program exits without calling `dglclose()` or terminates abnormally.

A non-zero exit status implies an abnormal exit. If the graphics server program detects a condition that is fatal, it exits with an *errno* value that best indicates the condition. If a system call or service returned an error number (*errno* or *h_errno*), this number is used as the exit number.

Table 19-3 lists all exit values that are internally generated (not the result of a failed system call or service).

Exit Value	Explanation
0	normal exit
ENODEV	invalid communication connection type
ENOMEM	out of memory
EINVAL	invalid command line argument
ETIMEDOUT	connection timed out
EACCESS	login incorrect or permission denied
EIO	read or write error
ENOENT	invalid Graphics Library routine number
ENOPROTOOPT	dgl/tcp service not found in <i>/etc/services</i>
ERANGE	invalid or unrecognizable number representation

Table 19-3 DGL Server Exit Value

Appendix A

Scope of GL Subroutines

This appendix lists all the GL subroutines and defines the scope of each subroutine. Each of the GL subroutines has a scope that determines how it affects system resources. Subroutines can affect the state of the currently selected framebuffer, the state of the current window, the state of the current process, or the state of the current graphics connection.

The state types listed in Table A-1 define the scope of GL subroutines. State types describe the system resource on which the subroutine operates.

State type	Description
Colormap	There is a separate screen-wide color map for each framebuffer
Display	A collection of screens and input devices
Framebuffer	A particular set of bitplanes; draw mode dependent
Graphics connection	Graphics client/server connection by <code>dglopen</code> or by default
mmode dependent	Depends on the current matrix mode
Obsolete	No longer supported, not recommended
Process	User's GL application
Renders	Renders into current framebuffer, which is selected by drawmode or affects a non-modal framebuffer state, such as texture coordinates, trimming curves
Screen	Collection of framebuffers, color maps, and video hardware
Textport	Affects a different process from the caller's window

Table A-1 GL State Types

Table A-2 lists the state type of each GL programming subroutine.

GL Subroutine	State Type	Comments
acbuf()	Window	
acsize()	Framebuffer	
addtopup()	Graphics connection	
afunction()	Window	
arc()*	Renders	
attachcursor()	Display	
backbuffer()	Framebuffer	Attribute
backface()	Window	
bbox2()*	Window	*Represents a family of subroutines
bgnclosedline()	Renders	
bgnline()	Renders	
bgnpoint()	Renders	
bgnpolygon()	Renders	
bgnqstrip()	Renders	
bgnsurface()	Renders	
bgntmesh()	Renders	
bgnttrim()	Renders	
blankscreen()	Screen	
blanktime()	Screen	
blendcolor()	Window	
blendfunction()	Window	
blink()	Colormap	
blkqread()	Graphics connection	

Table A-2 Scope of GL Subroutines

GL Subroutine	State Type	Comments
<code>c()*</code>	Framebuffer	Attribute, OK to call between <code>bgn/end</code> , *Represents a family of subroutines
<code>callfunc()</code>	Process	Not supported over <code>DGLTsocket</code> connections
<code>callobj()</code>	Graphics connection	
<code>charstr()</code>	Renders	
<code>chunksize()</code>	Graphics connection	
<code>circ()*</code>	Renders	*Represents a family of subroutines
<code>clear()</code>	Renders	
<code>clearhitcode()</code>	Window	
<code>clipplane()</code>	Window	
<code>clkoff()</code>	Display	
<code>clkon()</code>	Display	
<code>closeobj()</code>	Graphics connection	
<code>cmode()</code>	Framebuffer	Attribute, Takes effect when <code>gconfig()</code> is executed
<code>cmov()*</code>	Window	*Represents a family of subroutines
<code>color()*</code>	Framebuffer	Attribute, OK to call between <code>bgn/end</code> , *Represents a family of subroutines
<code>compactify()</code>	Graphics connection	
<code>concave()</code>	Window	
<code>cpack()</code>	Framebuffer	Attribute, OK to call between <code>bgn/end</code>
<code>crv()</code>	Renders	
<code>crvn()</code>	Renders	
<code>curorigin()</code>	Graphics connection	

Table A-2 (continued) Scope of GL Subroutines

GL Subroutine	State Type	Comments
<code>cursoff()</code>	Window	
<code>curson()</code>	Window	
<code>curstype()</code>	Graphics connection	
<code>curvebasis()</code>	Window	
<code>curveit()</code>	Renders	
<code>curveprecision()</code>	Window	
<code>cyclemap()</code>	Colormap	
<code>czclear()</code>	Renders	
<code>dbtext()</code>	Display	
<code>defbasis()</code>	Graphics connection	
<code>defcursor()</code>	Graphics connection	
<code>deflfont()</code>	Graphics connection	
<code>deflinestyle()</code>	Graphics connection	
<code>defpattern()</code>	Graphics connection	
<code>defpup()</code>	Graphics connection	
<code>defrasterfont()</code>	Graphics connection	
<code>delobj()</code>	Graphics connection	
<code>deltag()</code>	Graphics connection	
<code>depthcue()</code>	Window	
<code>dglclose()</code>	Process	
<code>dglopen()</code>	Process	
<code>dither()</code>	Window	
<code>dopup()</code>	Graphics connection	
<code>doublebuffer()</code>	Framebuffer	Takes effect when <code>gconfig()</code> is executed

Table A-2 (continued) Scope of GL Subroutines

GL Subroutine	State Type	Comments
<code>draw()</code> *	Renders	*Represents a family of subroutines
<code>drawmode()</code>	Window	Attribute
<code>editobj()</code>	Graphics connection	
<code>endclosedline()</code>	Renders	
<code>endcurve()</code>	Renders	
<code>endfeedback()</code>	Window	
<code>endfullscrn()</code>	Renders	
<code>endline()</code>	Renders	
<code>endpick()</code>	Window	
<code>endpoint()</code>	Renders	
<code>endpolygon()</code>	Renders	
<code>endpupmode()</code>	Obsolete	
<code>endqstrip()</code>	Renders	
<code>endselect()</code>	Window	
<code>endsurface()</code>	Renders	
<code>endtmesh()</code>	Renders	
<code>endtrim()</code>	Renders	
<code>feedback()</code>	Window	
<code>finish()</code>	Window	
<code>fogvertex()</code>	Window	
<code>font()</code>	Window	Attribute
<code>foreground()</code>	Process	Applies to next <code>winopen()</code> , <code>swinopen()</code> , <code>winconstraints()</code> call
<code>freepup()</code>	Graphics connection	

Table A-2 (continued) Scope of GL Subroutines

GL Subroutine	State Type	Comments
<code>frontbuffer()</code>	Framebuffer	Attribute
<code>frontface()</code>	Window	
<code>fudge()</code>	Graphics connection	Applies to next <code>winopen()</code> , <code>swinopen()</code> , <code>winconstraints()</code> call
<code>fullscrn()</code>	Window	
<code>gammaramp()</code>	Screen	
<code>gbegin()</code>	Graphics connection	
<code>gconfig()</code>	Window	
<code>genobj()</code>	Graphics connection	
<code>gentag()</code>	Graphics connection	
<code>getbackface()</code>	Window	
<code>getbuffer()</code>	Framebuffer	
<code>getbutton()</code>	Display	
<code>getcmmode()</code>	Window	
<code>getcolor()</code>	Framebuffer	Attribute
<code>getcpos()</code>	Window	
<code>getcursor()</code>	Window	
<code>getdcm()</code>	Window	
<code>getdepth()</code>	Obsolete	
<code>getdescender()</code>	Window	Attribute
<code>getdev()</code>	Graphics connection	
<code>getdisplaymode()</code>	Framebuffer	Attribute
<code>getdrawmode()</code>	Window	
<code>getfont()</code>	Window	Attribute

Table A-2 (continued) Scope of GL Subroutines

GL Subroutine	State Type	Comments
getgconfig()	Window	
getgdesc()	Screen	
getgpos()	Window	
getheight()	Window	Attribute
gethitcode()	Window	
getlsbackup()	Window	Attribute
getlsrepeat()	Window	Attribute
getlstyle()	Window	Attribute
getlwidth()	Window	Attribute
getmap()	Framebuffer	
getmatrix()	mmode dependent	
getmcolor()	Colormap	
getmmode()	Window	
getmonitor()	Screen	
getmultisample()	Window	
getnurbsproperty()	Window	
getopenobj()	Graphics connection	
getorigin()	Window	
getothermonitor()	Obsolete	
getpattern()	Window	
getplanes()	Window	
getport()	Obsolete	
getresetls()	Window	
getscrbox()	Window	
getscrmask()	Window	

Table A-2 (continued) Scope of GL Subroutines

GL Subroutine	State Type	Comments
getshade()	Obsolete	
getsize()	Window	
getsm()	Window	Attribute
getvaluator()	Display	
getvideo()	Screen	
getviewport()	Window	
getwritemask()	Framebuffer	Attribute
getwscrn()	Window	
getzbuffer()	Framebuffer	
gexit()	Graphics connection	
gflush()	Window	
ginit()	Graphics connection	
glcompat()	See below	
GLC_OLDPOLYGON	Window	
GLC_ZRANGEMAP	Graphics connection	
glresources()		
glxchoosevisual()	Graphics connection	
glxgetconfig()	Graphics connection	
glxlink()	Graphics connection	
glxunlink()	Graphics connection	
glxwindone()	Graphics connection	
glxwinset()	Graphics connection	
greset()	Window	
gRGBcolor()	Framebuffer	Attribute
gRGBcursor()	Obsolete	

Table A-2 (continued) Scope of GL Subroutines

GL Subroutine	State Type	Comments
<code>gRGBmask()</code>	Framebuffer	Attribute
<code>gselect()</code>	Window	
<code>gsync()</code>	Window	
<code>gversion()</code>	Screen	
<code>iconsize()</code>	Graphics connection	Applies to next <code>winopen()</code> , <code>swinopen()</code> , <code>winconstraints()</code> call
<code>icontitle()</code>	Window	
<code>imakebackground()</code>	Graphics connection	Applies to next <code>winopen()</code> , <code>swinopen()</code> , <code>winconstraints()</code> call
<code>initnames()</code>	Window	
<code>ismex()</code>	Obsolete	
<code>isobj()</code>	Graphics connection	
<code>isqueued()</code>	Graphics connection	
<code>istag()</code>	Graphics connection	
<code>keepaspect()</code>	Graphics connection	Applies to next <code>winopen()</code> , <code>swinopen()</code> , <code>winconstraints()</code> call
<code>lampoff()</code>	Display	
<code>lampon()</code>	Display	
<code>lcharstr()</code>	Renders	
<code>leftbuffer()</code>	Framebuffer	Attribute
<code>linesmooth()</code>	Window	
<code>linewidth()</code>	Window	Attribute
<code>linewidthf()</code>	Window	

Table A-2 (continued) Scope of GL Subroutines

GL Subroutine	State Type	Comments
<code>lmbind()</code>	See below	
<code>BACKMATERIAL</code>	Window	
<code>MATERIAL</code>	Window	OK to call between bgn/end
<code>LMODEL</code>	Window	
<code>LIGHT</code>	Window	
<code>lmcolor()</code>	Window	OK to call between bgn/end
<code>lmdef()</code>	See below	
<code>DEFMATERIAL</code>	Graphics connection	
<code>DEFMODEL</code>	Graphics connection	
<code>DEFLIGHT</code>	Graphics connection	
<code>loadmatrix()</code>	mmode dependent	
<code>loadname()</code>	Window	
<code>logicop()</code>	Window	
<code>lookat()</code>	mmode dependent	
<code>lrectread()</code>	Window	
<code>lrectwrite()</code>	Window	
<code>lRGBrange()</code>	Window	
<code>lsbackup()</code>	Window	
<code>lsetdepth()</code>	Window	
<code>lshaderange()</code>	Window	
<code>lsrepeat()</code>	Window	Attribute
<code>lstrwidth()</code>	Window	Attribute
<code>makeobj()</code>	Graphics connection	
<code>maketag()</code>	Graphics connection	

Table A-2 (continued) Scope of GL Subroutines

GL Subroutine	State Type	Comments
<code>mapcolor()</code>	Colormap	There is a separate screen-wide color map for each framebuffer
<code>mapw()</code>	Window	
<code>mapw2()</code>	Window	
<code>maxsize()</code>	Graphics connection	Applies to next <code>winopen()</code> , <code>swinopen()</code> , <code>winconstraints()</code> call
<code>minsize()</code>	Graphics connection	Applies to next <code>winopen()</code> , <code>swinopen()</code> , <code>winconstraints()</code> call
<code>mmode()</code>	Window	
<code>monobuffer()</code>	Framebuffer	Attribute
<code>move()*</code>	Renders	*Represents a family of subroutines
<code>msalpha()</code>	Window	
<code>msmask()</code>	Window	
<code>mspattern()</code>	Window	
<code>mssize()</code>	Framebuffer	
<code>mswapbuffers()</code>	Window	
<code>multimap()</code>	Framebuffer	Takes effect when <code>gconfig()</code> is executed
<code>multisample()</code>	Window	
<code>multimatrix()</code>	<code>mmode</code> dependent	
<code>n()*</code>	Window	*Represents a family of subroutines, OK to call between <code>bgn/end</code>
<code>newpup()</code>	Graphics connection	
<code>newtag()</code>	Graphics connection	
<code>nmode()</code>	Window	

Table A-2 (continued) Scope of GL Subroutines

GL Subroutine	State Type	Comments
noborder()	Graphics connection	Applies to next winopen(), swinopen(), winconstraints() call
noise()	Graphics connection	
noport()	Graphics connection	Applies to next winopen(), swinopen(), winconstraints() call
normal()	Obsolete	
nurbscurve()	Renders	
nurbssurface()	Renders	
objdelete()	Graphics connection	
objinsert()	Graphics connection	
objreplace()	Graphics connection	
onemap()	Framebuffer	Takes effect when gconfig() is executed
ortho()	mmode dependent	
ortho2()	mmode dependent	
overlay()	Window	Takes effect when gconfig() is executed
pagecolor()	Textport	
passthrough()	Window	
patch()	Renders	
patchbasis()	Window	
patchcurves()	Window	
patchprecision()	Window	
pclos()	Renders	
pdr()*	Renders	*Represents a family of subroutines

Table A-2 (continued) Scope of GL Subroutines

GL Subroutine	State Type	Comments
<code>perspective()</code>	mmode dependent	
<code>pick()</code>	Window	
<code>picksize()</code>	Window	
<code>pixmapmode()</code>	Window	
<code>pmv()*</code>	Renders	*Represents a family of subroutines
<code>pnt()*</code>	Renders	*Represents a family of subroutines
<code>pntsize()</code>	Renders	
<code>pntsizef()</code>	Renders	
<code>pntsmooth()</code>	Window	
<code>polarview()</code>	mmode dependent	
<code>polf()*</code>	Renders	*Represents a family of subroutines
<code>poly()*</code>	Renders	*Represents a family of subroutines
<code>polymode()</code>	Window	
<code>polysmooth()</code>	Window	
<code>popattributes()</code>	Window	
<code>popmatrix()</code>	mmode dependent	
<code>popname()</code>	Window	
<code>popviewport()</code>	Window	
<code>prefposition()</code>	Graphics connection	Applies to next <code>winopen()</code> , <code>swinopen()</code> , <code>winconstraints()</code> call
<code>prefsize()</code>	Graphics connection	Applies to next <code>winopen()</code> , <code>swinopen()</code> , <code>winconstraints()</code> call
<code>pupmode()</code>	Obsolete	
<code>pushattributes()</code>	Window	

Table A-2 (continued) Scope of GL Subroutines

GL Subroutine	State Type	Comments
pushmatrix()	mmode dependent	
pushname()	Window	
pushviewport()	Window	
pwlcurve()	Renders	
qcontrol()	Display	
qdevice()	Graphics connection	
qenter()	Graphics connection	
qgetfd()	Graphics connection	
qread()	Graphics connection	
qreset()	Graphics connection	
qtest()	Graphics connection	
rcrv()*	Renders	*Represents a family of subroutines
rdr()*	Renders	*Represents a family of subroutines
readdisplay()	Graphics connection	
readpixels()	Renders	
readRGB()	Renders	
readsource()	Framebuffer	
rect()*	Renders	*Represents a family of subroutines
rectcopy()	Renders	
rectread()	Renders	
rectwrite()	Renders	
rectzoom()	Renders	
resetls()	Window	
reshapeviewport()	Window	

Table A-2 (continued) Scope of GL Subroutines

GL Subroutine	State Type	Comments
<code>RGBcolor()</code>	Framebuffer	Attribute, OK to call between bgn/end
<code>RGBcursor()</code>	Obsolete	
<code>RGBmode()</code>	Framebuffer	OK to call between bgn/end, Attribute
<code>RGBrange()</code>	Obsolete	
<code>RGBwritemask()</code>	Framebuffer	Attribute
<code>ringbell()</code>	Display	
<code>rmv()*</code>	Renders	*Represents a family of subroutines
<code>rot()</code>	mmode dependent	
<code>rotate()</code>	mmode dependent	
<code>rpatch()</code>	Renders	
<code>rpdr()*</code>	Renders	*Represents a family of subroutines
<code>rpmv()*</code>	Renders	*Represents a family of subroutines
<code>sbox()*</code>	Renders	*Represents a family of subroutines
<code>scale()</code>	mmode dependent	
<code>sclear()</code>	Framebuffer	
<code>scrbox()</code>	Window	
<code>screenspace()</code>	Window	
<code>scrmask()</code>	Window	
<code>scrnattach()</code>	Display	
<code>scrnselect()</code>	Graphics connection	
<code>scrsubdivide()</code>	Graphics connection	
<code>setbell()</code>	Window	
<code>setcursor()</code>	Window	
<code>setdblights()</code>	Display	

Table A-2 (continued) Scope of GL Subroutines

GL Subroutine	State Type	Comments
setdepth()	Obsolete	
setlinestyle()	Window	Attribute
setmap()	Framebuffer	
setmonitor()	Screen	
setnurbsproperty()	Window	
setpattern()	Window	Attribute
setpup()	Graphics connection	
setshade()	Obsolete	
setvaluator()	Display	
setvideo()	Screen	
shademodel()	Window	Attribute
shaderange()	Obsolete	
singlebuffer()	Framebuffer	Takes effect when gconfig() is executed
smoothline()	Obsolete	
spclos()	Obsolete	
splf()*	Renders	*Represents a family of subroutines
stencil()	Framebuffer	
stensize()	Framebuffer	
stepunit()	Graphics connection	Applies to next winopen(), swinopen(), winconstraints() call
stereobuffer()	Framebuffer	Attribute
strwidth()	Window	Attribute
subpixel()	Window	
swapbuffers()	Window	

Table A-2 (continued) Scope of GL Subroutines

GL Subroutine	State Type	Comments
<code>swapinterval()</code>	Window	
<code>swaptmesh()</code>	Window	
<code>swinopen()</code>	Window	
<code>swritemask()</code>	Framebuffer	
<code>t()*</code>	Window	*Represents a family of subroutines
<code>tevbind()</code>	Window	
<code>tevdef()</code>	Graphics connection	
<code>texbind()</code>	Window	
<code>texdef2d()</code>	Graphics connection	
<code>texdef3d()</code>	Graphics connection	
<code>texgen()</code>	Window	
<code>textcolor()</code>	Textport	
<code>textinit()</code>	Textport	
<code>textport()</code>	Textport	
<code>tie()</code>	Graphics connection	
<code>tlutbind()</code>	Window	
<code>tpoff()</code>	Textport	
<code>tpon()</code>	Textport	
<code>translate()</code>	mmode dependent	
<code>underlay()</code>	Window	Takes effect when <code>gconfig()</code> is executed
<code>unqdevice()</code>	Graphics connection	
<code>v()*</code>	Renders	*Represents a family of subroutines
<code>videocmd()</code>	Screen	
<code>viewport()</code>	Window	

Table A-2 (continued) Scope of GL Subroutines

GL Subroutine	State Type	Comments
winattach()	Obsolete	
winclose()	Window	
winconstraints()	Window	
windepth()	Window	
window()	mmode dependent	
winget()	Window	
winmove()	Window	
winopen()	Window	
winpop()	Window	
winposition()	Window	
winpush()	Window	
winset()	Window	
wintitle()	Window	
wmpack()	Framebuffer	Attribute
writemask()	Framebuffer	Attribute
writepixels()	Renders	
writeRGB()	Renders	
xfpt()*	Renders	*Represents a family of subroutines
zbsize()	Framebuffer	
zbuffer()	Framebuffer	
zclear()	Framebuffer	
zdraw()	Framebuffer	
zfunction()	Framebuffer	
zsource()	Framebuffer	
zwritemask()	Framebuffer	

Table A-2 (continued) Scope of GL Subroutines

Appendix B

Global State Attributes

This appendix lists the initial values and the defaults for the GL global state attributes.

Table B-1 lists the default color map values.

Index	Name	RGB Values		
		Red	Green	Blue
0	BLACK	0	0	0
1	RED	255	0	0
2	GREEN	0	255	0
3	YELLOW	255	255	0
4	BLUE	0	0	255
5	MAGENTA	255	0	255
6	CYAN	0	255	255
7	WHITE	255	255	255
All others	Unnamed	Unchanged	Unchanged	Unchanged

Table B-1 Default Color Map Values

Table B-2 lists the keys to the supplemental information in Table B-3.

Key	Description
A	Is pushed and popped on the attributes stack
G	Takes effect when <code>gconfig()</code> is called
V	Can be changed between <code>bgn</code> and end calls: <code>bgnpoint()</code> , <code>bgnline()</code> , <code>bgnclosedline()</code> , <code>bgnpolygon()</code> , <code>bgnqstrip()</code> and <code>bgntmesh()</code>

Table B-2 Keys to Information in Table B-3

Table B-3 lists the global state attributes and their defaults.

Attribute	Initial Value	Key
<code>acsize()</code>	0	G
<code>afunction()</code>	0, AF_ALWAYS	
<code>backbuffer()</code>	FALSE	A
<code>backface()</code>	FALSE	
<code>blendfunction()</code>	BF_ONE, BF_ZERO	
character position	Undefined	
<code>clipplane()</code>	CP_OFF	
<code>cmode()</code>	TRUE	A, G
<code>color()</code>	0	A, V
<code>concave()</code>	FALSE	
<code>curveprecision()</code>	Undefined	
depth range	<code>getgdesc(GD_ZMIN)</code> , <code>getgdesc(GD_ZMAX)</code>	
<code>depthcue()</code>	FALSE	
<code>dither()</code>	DT_ON	
<code>doublebuffer()</code>	FALSE	G
<code>drawmode()</code>	NORMALDRAW	A

Table B-3 Global State Attribute Defaults

Attribute	Initial Value	Key
feedback mode	Off	
fogvertex()	FG_OFF	
font	0	A
frontbuffer()	TRUE	A
frontface()	FALSE	
full screen mode	Off	
glcompat()	See below	
GLC_OLDPOLYGON	1	
GLC_ZRANGEMAP	1 (B and G models) 0 (other models)	
graphics position	Undefined	
leftbuffer()	TRUE	
linesmooth()	SML_OFF	
linestyle()	0 (solid)	A
linewidth()	1	A
lmcolor()	LMC_COLOR	V
lmbind()	See below	
BACKMATERIAL	0	
LIGHT n	0	
LMODEL	0	
MATERIAL	0	
logicop()	LO_SRC	
lsrepeat()	1	A
mapcolor()	No entries changed	

Table B-3 (continued) Global State Attribute Defaults

Attribute	Initial Value	Key
matrix	See below	
ModelView	Undefined	
Projection	Undefined	
Single	ortho2 matching window size	
Texture	Undefined	
mmode()	MSINGLE	
monobuffer()	Enabled	
msalpha()	MSA_ALPHA()	
msmask()	1.0, 0	
mspattern()	MSP_DEFAULT	
mssize()	0	G
multimap()	FALSE	G
multisample()	TRUE	
name stack	Empty	
nmode()	NAUTO	
normal vector	Undefined	V
onemap()	TRUE	G
overlay()	2	G
patchbasis()	Undefined	
patchcurves()	Undefined	
patchprecision()	Undefined	
pattern	0 (solid)	A
pick mode	Off	
picksz()	10x10	
pixmap()	Standard	

Table B-3 (continued) Global State Attribute Defaults

Attribute	Initial Value	Key
<code>pntsmooth()</code>	<code>SMP_OFF</code>	
<code>polymode()</code>	<code>PYM_FILL</code>	
<code>polysmooth()</code>	<code>PYSM_OFF</code>	
<code>readsource()</code>	<code>SRC_AUTO</code>	
<code>rectzoom()</code>	1.0,1.0	
<code>rightbuffer()</code>	<code>FALSE</code>	
RGB color	All components 0 when RGBmode is entered	
RGB shade range	Undefined	
<code>RGBsize()</code>	12	
<code>RGBmode()</code>	<code>FALSE</code>	G A
RGB writemask	0xFF when RGB is entered	A,V
<code>scrbox()</code>	<code>SB_RESET</code>	
<code>scrmask()</code>	Set to size of window	
<code>scrsubdivide()</code>	<code>SS_OFF</code>	
select mode	Off	
shade range	0,7, <code>getgdesc(GD_ZMIN)</code> , <code>getgdesc(GD_ZMAX)</code>	
<code>shademodel()</code>	<code>GOURAUD</code>	A
<code>singlebuffer()</code>	<code>TRUE</code>	G
<code>stencil()</code>	Disabled	
<code>stensize()</code>	0	G
<code>stereobuffer()</code>	Disabled	
<code>swritemask()</code>	All stencil planes enabled	
<code>tevbind()</code>	0 (off)	
<code>texbind()</code>	0 (off)	
<code>texgen()</code>	<code>TG_OFF</code>	

Table B-3 (continued) Global State Attribute Defaults

Attribute	Initial Value	Key
<code>underlay()</code>	0	G
<code>viewport()</code>	Set to size of window	
<code>writemask()</code>	All bitplanes enabled	A
<code>zbsize()</code>	32	
<code>zbuffer()</code>	FALSE	
<code>zdraw()</code>	FALSE	A
<code>zfunction()</code>	ZF_EQUAL	
<code>zsource()</code>	ZSRC_DEPTH	
<code>zwritemask()</code>	All z-buffer planes enabled	

Table B-3 (continued) Global State Attribute Defaults

Appendix C

Transformation Matrices

Transformation commands create the following matrices.

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

$\text{Polarview}(\text{dist}, \text{azim}, \text{inc}, \text{twist}) = \text{Rot}_z(-\text{azim}) \cdot \text{Rot}_x(-\text{inc}) \cdot \text{Rot}_z(-\text{twist}) \cdot \text{Trans}(0.0, 0.0, -\text{dist})$
 $\text{lookat}(v_x, v_y, v_z, p_x, p_y, p_z, \text{twist}) = \text{trans}(-v_x, -v_y, -v_z) \cdot \text{rot}_y(\theta) \cdot \text{rot}_x(\phi) \cdot \text{rot}_z(-\text{twist})$

$$\sin \theta = \frac{p_x - v_x}{\sqrt{(p_x - v_x)^2 + (p_z - v_z)^2}}$$

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

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

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

C.5 Perspective Transformations

$$\text{perspective}(\text{fov}, \text{aspect}, \text{near}, \text{far}) = \begin{bmatrix} \frac{\cot\left(\frac{\text{fov}}{20}\right)}{\text{aspect}} & 0 & 0 & 0 \\ 0 & \cot\left(\frac{\text{fov}}{20}\right) & 0 & 0 \\ 0 & 0 & \frac{\text{far} + \text{near}}{\text{far} - \text{near}} & -1 \\ 0 & 0 & \frac{2 \cdot \text{far} \cdot \text{near}}{\text{far} - \text{near}} & 0 \end{bmatrix}$$

$$\text{window}(\text{left}, \text{right}, \text{bottom}, \text{top}, \text{near}, \text{far}) = \begin{bmatrix} \frac{2 \cdot \text{near}}{\text{right} - \text{left}} & 0 & 0 & 0 \\ 0 & \frac{2 \cdot \text{near}}{\text{top} - \text{bottom}} & 0 & 0 \\ \frac{\text{right} + \text{left}}{\text{right} - \text{left}} & \frac{\text{top} + \text{bottom}}{\text{top} - \text{bottom}} & \frac{\text{far} + \text{near}}{\text{far} - \text{near}} & -1 \\ 0 & 0 & \frac{2 \cdot \text{far} \cdot \text{near}}{\text{far} - \text{near}} & 0 \end{bmatrix}$$

C.6 Orthographic Transformations

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

$$\text{ortho2}(\text{left}, \text{right}, \text{bottom}, \text{top}) = \begin{bmatrix} \frac{2}{\text{right} - \text{left}} & 0 & 0 & 0 \\ 0 & \frac{2}{\text{top} - \text{bottom}} & 0 & 0 \\ 0 & 0 & -1 & 0 \\ \frac{\text{right} + \text{left}}{\text{right} - \text{left}} & \frac{\text{top} + \text{bottom}}{\text{top} - \text{bottom}} & 0 & 1 \end{bmatrix}$$

Appendix D

GL Error Messages

This appendix lists of error messages used in the Graphics Library. Errors are listed by the error number that is output by the system when your program has an error. A probable cause is listed for each error, except for errors that are listed as “not documented”.

This information is intended to be a suggestion about what you should consider when debugging your program. The cause listed may not be the reason you received that particular error message, but you should check your program for the listed condition before proceeding with debugging.

No.	Error	Probable Cause
1	ERR_SINGMATRIX	The matrix given cannot be inverted. Usually caused by scaling a zero value, or setting near or far planes to 0, or to equal values.
2	ERR_OUTMEM	Ran out of memory, <i>malloc</i> failed.
3	ERR_NEGSIDES	Program attempted to generate a polygon with less than zero sides.
4	ERR_BADWINDOW	Values for <i>left-right</i> , <i>top-bottom</i> , or <i>far-near</i> were probably switched. The <i>left</i> , <i>top</i> , and <i>far</i> values must be greater than <i>right</i> , <i>bottom</i> , and <i>near</i> values.
5	ERR_NOOPENOBJ	Program attempted to edit an object when there was no object to edit.
6	ERR_NOFONTRAM	Obsolete.

Table D-1 GL Error Messages and Probable Causes

No.	Error	Probable Cause
7	ERR_FOV	<i>fovy</i> must be greater than 1.0 and less than or equal to 1800. Note: the <i>fovy</i> should be ten times the angle than is the result of the tangent of the window width and the distance from viewer to screen for best results, that is, <i>fovy</i> for a full screen seen 2 feet away should be about 337 (inverse tan of 16/24).
8	ERR_BASISID	Pointers to the matrices being used for precision or curve type bspline, cardinal, or bezier are not equivalent or the pointer is null. Also, possibly one of the basis id's was not found, an undefined basis number is used.
9	ERR_NEGINDEX	Not documented.
10	ERR_NOCLIPPERS	Does not occur on IRIS 4D systems
11	ERR_STRINGBUG	Not documented.
12	ERR_NOCURVBASIS	Basis for the curve must be set before drawing the curve.
13	ERR_BADCURVID	Not documented.
14	ERR_NOPTCHBASIS	Basis for the patch must be set before drawing the patch.
15	ERR_FEEDPICK	Not documented.
16	ERR_INPICK	Program is already picking, can't try to pick twice at same time.
17	ERR_NOTINPICK	Program not in pick mode when trying to end feedback or end pick or select mode.
18	ERR_ZEROPICK	The size of the picking region is non-positive.
19	ERR_FONTBUG	Not documented.
20	ERR_INRGB	Program attempted to set mapcolor or writemask while in RGBmode.

Table D-1 (continued) GL Error Messages and Probable Causes

No.	Error	Probable Cause
21	ERR_NOTINRGBMODE	Program attempted to use a RGBmode routine while not in RGBmode.
22	ERR_BADINDEX	Color is not within valid index range.
23	ERR_BADVALUATOR	Program attempted to use a non-existing valuator.
24	ERR_BADBUTTON	Program attempted to access an illegal button number.
25	ERR_NOTINDBMODE	Program should be in dial button box mode to call current procedure.
26	ERR_BADINDEXBUG	Not documented.
27	ERR_ZEROVIEWPORT	The values for {x1 y1} are larger than {x2 y2} in the definition of screen coordinates. Thus, the size of the viewport is too small.
28	ERR_DIALBUG	Not Documented.
29	ERR_MOUSEBUG	Not Documented.
30	ERR_RETRACEBUG	Not Documented.
31	ERR_MAXRETRACE	Program attempted to execute more than the maximum number of retrace events allowed.
32	ERR_NOSUCHTAG	The tag chosen is not in the hash table or at the beginning of the object set up. It cannot be found.
33	ERR_DELBUG	Not documented.
34	ERR_DELTAG	Program attempted to delete a non-existent or incorrect tag.
35	ERR_NEGTAG	Object tag is less than zero.
36	ERR_TAGEXISTS	Program attempted to create an object with an index number that already exists.

Table D-1 (continued) GL Error Messages and Probable Causes

No.	Error	Probable Cause
37	ERR_OFFTOOBIG	Offset is too large, program attempted to step beyond edge of object.
38	ERR_ILLEGALID	Program attempted to make an object with an id that is negative.
39	ERR_GECONVERT	Not documented.
40	ERR_BADAXIS	Axis must be named with {x X y Y z Z} anything else will give error.
42	ERR_BADDEVICE	The graphics manager couldn't be reached. This is a fatal error.
44	ERR_PATCURVES	Not documented.
45	ERR_PATPREC	Not documented.
46	ERR_CURVPREC	Not documented.
47	ERR_PUSHATTR	The attribute stack is full, pop or clear attribute stack first. check for unbalanced code.
48	ERR_POPATTR	The attribute stack is empty, something must be put there first. Check for unbalanced code.
49	ERR_PUSHMATRIX	The matrix stack is full. Clear or pop first. Check for unbalanced code.
50	ERR_POPMATRIX	The matrix stack is empty, something must be put there first. check for unbalanced code.
51	ERR_PUSHVIEWPORT	The viewport stack is full, pop or clear viewport stack first. Check for unbalanced code.
52	ERR_POPVIEWPORT	The viewport stack is empty, something must be put there first. Check for unbalanced code.
53	ERR_SIZEFIXED	Object size is frozen, chunksize cannot change after start of object definition.

Table D-1 (continued) GL Error Messages and Probable Causes

No.	Error	Probable Cause
54	ERR_SETMONITOR	Not documented. {illegal monitor type?}
55	ERR_CHANGEINDEX0	Program was attempting to redefine the default style of {line cursor font etc}, change index 0 to some other value.
56	ERR_BADPATTERN	Bad size of pattern, only use 16, 32 or 64
67	ERR_CURSORNOTFOUND	An invalid cursor name was given, check and change value.
58	ERR_FONTHOLES	Not documented.
59	ERR_REPLACE	Program attempted to replace past end of object.
60	ERR_STARTFEED	Not documented.
61	ERR_CYCLEMAP	Not documented.
62	ERR_TAGINREPLACE	Program attempted to make a tag in replace mode. only replace graphics commands
63	ERR_TOOFEWPTS	The Program attempted to create a rational curve with less than 4 points.
64	ERR_UNDEFINEDCHAR	Not documented.
65	ERR_BADCURSOR	Bad size, only use 16, or 32 or default cursor types.
66	ERR_NOTINCOLORMODE	To use <i>getcurs</i> , program must be in proper mode. Not documented.
67	ERR_UNKNOWNCMDLENGTH	Routine fell through the case statement for checking size of number of polygon points. New command must be same length as the old command.
68	ERR_INFEEDBACK	Program trying to enter feedback when in feedback mode.

Table D-1 (continued) GL Error Messages and Probable Causes

No.	Error	Probable Cause
69	ERR_DURINGFEEDBACK	Program trying to exit feedback when not in feedback.
70	ERR_DURINGSELECT	System failure occurred in <code>endpick</code> .
71	ERR_ARGMISMATCH	Buffer specified in <code>endfeedback</code> , <code>endpick</code> or <code>select</code> , must be same as buffer started with
72	ERR_TOOMANYARGS	Program attempted to use more than ten arguments in a function call. This is error.
73	ERR_OBJNOTFOUND	There is no header or no valid header in the object the Program called. object was invalid.
74	ERR_MAKEROOMINREPLACEMODE	Program attempted to reorganize display list entries while in replace mode. This is a error.
75	ERR_UNABLETOOPEN	The <i>gid</i> value given to the hardware is invalid or less than 0.
76	ERR_QUEUEINGDEVICE	Program attempted to queue a non-existing device.
77	ERR_UNQUEUEINGDEVICE	Program attempted to unqueue a non-existing device.
78	ERR_GETBUTTONERROR	Program attempted to get a non-existing button value.
79	ERR_GETVALUATORERROR	Program attempted to get an invalid valuator.
80	ERR_SETVALERROR	Program attempted to set an invalid valuator.
81	ERR_TIEERROR	Program attempted to tie a non-valid device id.
82	ERR_NOISEERROR	Not documented.
83	ERR_ATTACHCURSOR	Not documented.

Table D-1 (continued) GL Error Messages and Probable Causes

No.	Error	Probable Cause
84	ERR_MAPDEVICE	Not documented.
85	ERR_WINATTACH	Not documented.
86	ERR_NOSUCHWINDOW	Program attempted to switch to or close an invalid window.
87	ERR_CLOSEDLASTWINDOW	Not documented.
88	ERR_LINESTYLENOTFOUND	Not documented.
89	ERR_PATTERNNOTFOUND	Not documented.
90	ERR_NULLWSINCLONING	Not documented.
91	ERR_USERERROR	Error message, used for anything that won't fit in other messages
92	ERR_NOFONTFOUND	The name of the font given was not found, check the list in <i>/usr/lib/fmfonts</i>
93	ERR_WMANIPC	Message being sent to the window manager was too large.
94	ERR_INPUTOPEN	Not documented.
95	ERR_RESETINGQ	Program attempted to reset a non-valid queue id.
96	ERR_GETTP	Program attempted to access an non-valid textport id.
97	ERR_TOOMANYSIDES	Program attempted to create a polygon with more than 255 vertices.
98	ERR_INVALIDMODE	Program attempted to enter an invalid mode.
99	ERR_INVALIDPARENT	There is no parent state for current window.
106	ERR_NOWIN	No window manager running.

Table D-1 (continued) GL Error Messages and Probable Causes

Appendix E

Using Graphics and Share Groups

Graphics is a shared attribute among processes that share virtual address space by using the *sproc* system call. Prior to IRIX release 3.3, only a single thread of a share group could perform a `winopen()` and be allowed access to the graphics pipe. The flexibility of having any thread performing a graphics call may result in increased performance for those applications that take advantage of this feature, but there are a number of caveats that the programmer should be aware of. This appendix describes the potential problems and how to avoid them.

The graphics pipe on all architectures is a memory mapped device—that is, loads and stores to memory locations send commands and data to the hardware. Thus the graphics hardware is a region of virtual memory that is made accessible to a graphics process once it has done a `winopen()` call. The semantics of the `PR_SADDR` option of *sproc* are extended so that children created with this option share this virtual address space as well as sharing regular memory. Because access to other graphics resources (input queues, and so on) is done through open file descriptors, the `PR_SFDS` option should be enabled as well.

The resulting effect is that when one thread performs a graphics call, it is as if all threads perform the call. For example, if one process performs a `winopen()`, all threads in the share group have access to the new window. If any thread performs a `winset()`, any succeeding graphics call by any other thread is applied to the new window. A `color()` call sets the display color for all threads, and so on.

Unlike the library routines in *libmpc.a*, no code has been added to the IRIS GL to prevent simultaneous access by separate processes to either GL data structures or to the graphics pipe. Because the programming model presented by the GL is fundamentally modal, the responsibility for the definition and

protection of critical regions must be owned by the application program. For example, suppose that one process within the share group wants to perform the following sequence, each for a different polygon:

```
bgnpolygon(), v3f(), v3f(), ... endpolygon()
```

As soon as the process has made the `bgnpolygon()` call, any other process in the share group may not perform a GL call until the first process has performed the corresponding `endpolygon()` call. Thus, the application code must make the above sequence a critical region in each process, in order to ensure that the two processes do not interleave their sequences of calls. The routines described in *ussetlock* that use test-and-set style spinlocks are one effective way of enforcing the synchronization.

The effects of failing to synchronize access to the graphics pipe and associated data structures are unpredictable. At the least, some display anomalies will occur. The most catastrophic result is unexpected shutdown of the window manager and the graphics subsystem.

There are two other rules to follow when using graphics share groups. These are issues with the current implementation and may not apply to a future release. First, the process that performed the initial `winopen()` must remain alive while any thread performs GL calls. Second, unless a `foreground()` hint call is made prior to the `winopen()`, the `winopen()` call should happen prior to any call to *sproc*. This is because `winopen()` by default places a process in the background by calling *fork* and having the original parent process die. The *fork* will cause the new process to exit the share group. Delaying the *sproc* until after the `winopen()` creates the share group after graphics has been initialized.

This list summarizes the rules for sharing graphics among processes:

1. Perform a *sproc* call with sharing options `PR_SADDR` and `PR_SFDS`.
2. Treat all atomic sequences of GL calls as critical regions.
3. The process that did the first `winopen()` must not exit until all threads are finished performing GL calls.
4. Perform the *sproc* call after the `winopen()` call unless the `foreground()` call is used.

Index

A

- acbuf, 15-31
- accumulation buffer, 15-30 through 15-36
- acsize, 15-30
- addressing
 - frame buffer memory, 6-1
 - frame buffers, 10-1
 - pixels, 11-4
- afunction, 8-22, 18-45
- algebraic representations, 14-2
- aliasing, definition of, 15-1
- ALPHA, 9-21
- alpha
 - bitplanes, 15-7
 - blending, 8-22 through 8-26
 - hints for texture, 18-45
 - of material, 9-21
- AMBIENT, 9-8, 9-9
- ambient light, 9-4, 9-9
- ambient property, fast update of, 9-20
- animation, 6-1 through 6-8
 - flicker, 6-2
 - frame rate, 6-3
 - gsync not recommended, 6-8
 - maximizing performance, 6-6
 - setting swap interval for, 6-7
 - setting up, 6-3
- anomalies, display, 2-20
- ANSI C, 1-1, 1-4
 - cckr non-ANSI compatibility flag, 1-4
- antialiasing, 15-1 through 15-36
 - box filter, 15-35
 - definition, 15-1
 - Gaussian filter, 15-35
 - lines, 15-15
 - multipass, 15-30 through 15-36
 - onepass, 15-10 through 15-30
 - on RealityEngine, 15-36 through 15-47
 - polygons, 15-22
 - RGB lines caution, 15-22
 - RGB mode, 15-13
 - subpixel, 15-4 through 15-5
- approximating surfaces
 - non-planar polygons, 2-16
- apropos*, xxv
- arc/arcf, 2-36
- arcs, 2-35
- aspect ratio, 7-5
 - maintaining, 7-6
- asymmetric viewing volume, 7-9, 7-11
- Athena widgets, 5-2
- atmospheric effects, 13-10 through 13-14
- attachcursor, 5-6
- ATTENUATION, 9-15
- ATTENUATION2, 9-15

attenuation of a spotlight, 9-17
attenuation of light, 9-14
azimuth, 7-13

B

B-Spline, 14-6
 interpolate control points, 14-10
 old-style curve, 14-29
 relationships, 14-9
back buffer, 6-1, 6-6
backbuffer, 6-2
 and zdraw, 8-10
backface, 8-22
backface removal, 8-21 through 8-22
backfacing polygons, 8-21
BACKMATERIAL, 9-18, 9-20
basis functions, 14-7, 14-8
basis matrix, 14-28
bbox2, 16-7
Bezier cubic curve, 14-29
bgnclosedline/endclosedline, 2-9
bgncurve/endcurve, 14-11
bgncurve/endcurve, 14-11
bgnline/endline, 2-8
bgnpoint/endpoint, 2-7
bgnpolygon/endpolygon, 2-15
bgnqstrip/endqstrip, 2-27
bgnsurface/endsurface, 14-13
bgntmesh/endtmesh structure, 2-21
bgnttrim/endtrim structure, 14-20
bicubic patch, 14-33
binary compatibility, 1-4
binding lighting, 9-11
bitmask, 3-7, 3-8
bitplane
 selecting for drawing, 10-1
 writemasks, 10-7

bitplanes, 4-15, 6-2, 8-3
 allocating for accumulation, 15-30
 color, 4-3
 configuring, 10-15 through 10-16
 definition, 10-1
 overlay, 10-3
 underlay, 10-3
bits per pixel, 4-3, 4-5
blankscreen, 5-18
blanktime, 5-18
blendfunction, 15-5
blendfunction, 15-4
blending, 15-5 through 15-9
 correlated, 15-26
 texture colors, 18-41
blinking, 4-21
blkqread, 5-5
blurring effect of texture MipMap, 18-21
bounding box, 16-7
bowtie polygon, 2-15
 rendering, 2-16
box filter, 15-35
buffer
 picking, 12-2
 querying mode of, 6-7
 writing to back, 6-6
buffers
 swapping, 6-3
 swapping multiple, 6-3
button
 polling, 5-8
 reading, 5-6
buttons, 5-11
byte stream, 19-2

C

- cc*, 1-4
- cckr**, 1-4
- c3s*, 4-7
- caching of texture memory, 18-45
- calculations, precision of, 2-16
- callbacks limited for remote operation, 19-5
- callfunc*, 19-5
- callobj*, 16-4
- Cardinal spline curve, 14-29
- cartesian floating point coordinates, 7-2
- C compiler, 1-4
- changing currently bound definitions, 9-13
- changing defined objects, 16-8
- changing lighting settings, 9-12 through 9-13
- changing pixel fill directions, 11-12
- changing the color map, 4-16
- character, 3-2
 - bitmasks, 3-8
 - clipping, 3-4
 - transformations, 3-6
- charstr*, 3-2
- choosing items on the screen, 12-1
- choosing parameters for *ortho*, 7-11
- chunk, 16-13
- chunksize*, 16-12
- circ/circonf*, 2-34
- circle, representing with NURBS, 14-12
- circles, 2-34
- clearing stencil planes, 8-19
- client/server model, 19-1
- clip coordinates, 7-3
- clipping
 - characters, 3-4
 - effect on feedback, 17-3
 - gross, 3-4
- clipping planes, 7-5, 7-8, 7-9, 8-7
 - user-defined, 7-30
- clipplane*, 7-30
- clkoff*, 5-16
- clkon*, 5-16
- closed line, definition, 2-8
- closeobj*, 16-2, 16-8
- closing an object definition, 16-2
- closing a window, 5-14
- cmov*, 3-2
- coefficients, 14-5
- color, 4-1 through 4-25
 - blending, 15-5 through 15-9
 - how monitors display, 4-2
 - how the GL calculates for lighting, 9-6
 - in lighted scenes, 9-2
 - mode, 4-3
 - NURBS surface, 14-13, 14-17
 - smooth variation, 4-9
- color*, 4-17, 10-5
- color data, 8-bit, 4-5
- color display correction, 4-23
- colorf*, 4-17
- color guns, 4-2
- color index, 4-17
- color index mode, see color map mode
- color map
 - assigning in multimap mode, 4-22
 - changing, 4-16
 - default, 4-16
 - getting index of in multimap mode, 4-22
 - getting values of, 4-18
- color map mode, 4-15 through 4-20
 - antialiased points, 15-11
 - depth-cueing, 13-4
 - lighting, 9-24
 - querying, 4-22
- color mode
 - querying, 4-8

- color ramp, 13-5
- compactify, 16-12
- compatibility, 1-9
 - GLC_OLDPOLYGON, 2-31
- compiling C programs, 1-4
- concave, 2-13
- concave polygon, 2-13
- configuration, remote, 19-12
- configuring overlay/underlay
 - bitplanes, 10-15 through 10-17
- conic sections, representing
 - with NURBS, 14-12
- connection, 19-4
 - default, 19-4
 - permission, 19-4
 - TCP, 19-12
- contour texture, 18-35
- control net, 14-13
- control points, 14-5, 14-6
 - agree with data type, 14-15
 - size of array, 14-14
- control polygon, 14-6
- convex polygon, 2-12
- coordinates
 - cartesian, 7-2
 - clip, 7-3
 - eye, 7-3
 - normalized, 7-3
 - object, 7-2
 - window, 7-3
 - world, 7-3
- coordinate systems, 7-2 through 7-3
- copying pixels, 11-7
- correlated blending function, 15-26
- cpack, 4-7
- C programming language, xxi, 1-4
- creating a cursor, 10-19
- creating a font, 3-7 through 3-10
- creating a glowing surface, 9-4
- creating an object, 16-2
- creating highlights with lighting, 9-3
- creating queue entries, 5-4
- cross-hair cursor, 10-17
- cross-plot, 14-4
- cubic polynomials, 14-5
- culling
 - definition, 16-5
 - NURBS, 14-21
- curorigin, 10-20
- current character position, 3-2
 - finding, 3-4
- current graphics position, 2-37
- current matrix, 7-16
- current normal, 9-5
- current texture matrix, 18-33
- cursoff, 5-6
- curson, 5-6
- cursor, 5-6
 - creating, 10-19
 - cross-hair, 10-17
 - default, 10-17
 - defining a glyph, 10-20
 - defining the type, 10-20
 - finding, 5-7
 - hardware, 10-17
 - mask, 10-17
 - mode, 10-6
 - origin, 10-20
 - turning on/off, 5-6
 - types, 10-17
- cursor devices, 5-13
- CURSORDRAW, 10-6
- cursors, 10-17 through 10-23
- curstype, 10-20
- curves
 - old-style, 14-28, 14-28 through 14-33
- cyclemap, 4-22
- czclear, 8-8, 8-9

D

daemon, 19-12

dgl, 19-13

inetd, 19-13

dbtext, 5-17

decals textures, 18-42

default connection for remote operation, 19-4

default cursor, 10-17

default settings for lighting, 9-13

defcursor, 10-20

defining a screen region for selecting, 12-8

defining a cursor glyph, 10-20

defining a material, 9-8

defining a viewing point, 7-13

deflinestyle, 2-10

defpattern, 2-16

defrasterfont, 3-8

degree elevation of polynomials, 14-5

deleting an object, 16-4

delobj, 16-4

deltag, 16-10

demos

NURBS Curve, 14-1

Trimmed NURBS, 14-1

density of fog, 13-12

dependent variable, 14-2

depth-cueing, 13-1 through 13-10

 in color map mode, 13-4

 optimizing resolution for, 7-5

depth-cueing in RGB mode, 13-6

DEPTHCHANGE, 5-14

depthcue, 13-4

depth perception, 13-1

DetailTexture, 18-27 through 18-32

determining the maximum z value, 8-7

determining the minimum z value, 8-7

device

 disabling queuing, 5-4

 finding out if queued, 5-4

 initialization, 5-16

 queueing, 5-4

 Spaceball, 5-15

device domain, 5-10

devices, 5-10 through 5-17

 ghost, 5-13

 labeling dial and button box, 5-17

DGL client, 19-2

dgl daemon, 19-13

DGL protocol, 19-2, 19-12

DGLSERVER, 19-4

DGL server, 19-2

DGLTsocket, 19-4

DGLTYPE, 19-4

dial and button box, 5-17

difference image, 18-28

DIFFUSE, 9-8

diffuse reflectance, 9-3

diffuse reflection, fast update, 9-20

disabling queueing for a device, 5-4

discontinuity, 14-8

DISPLAY, 1-2, 1-3, 19-4

display

 getting mode, 6-7

 hostname, 1-3

display anomalies, 2-20

displaying graphics over the network,
 19-1 through 19-16

display lists, 16-1

distance-squared attenuation, 9-15

distortion

 from non-planar polygons, 2-16

 in color displays, 4-23

dither, 4-4

dithering, 4-4

- doublebuffer, 6-2
- double buffering, 6-1
- double buffer mode, 6-1, 6-2
- draw, 2-39
- drawing, 2-1 through 2-37, 3-2
 - arcs, 2-35
 - characters, 3-2
 - circles, 2-34
 - cursors, 10-17 through 10-23
 - meshes, 2-21 through 2-28
 - mode, 10-5 through 10-7
 - object in display list, 16-4
 - old-style, 2-30 through 2-42
 - on top of pixels, 10-1
 - points, 2-7
 - polygons, 2-15
 - polylines, 2-8
 - rectangles, 2-32
 - setting bitplane destination, 10-5
 - underneath pixels, 10-1
- drawing mode, 10-1
- drawmode, 10-6
 - pixel access, 11-4
- driving a plotter with feedback, 17-2

E

- 8-bit color data, 4-5
- editobj, 16-8
- effect of clipping on feedback, 17-3
- electrons, 4-2
- emission, 9-4
- EMMISSION, 9-9
- emptying the event queue, 5-3
- enabling queueing for a device, 5-4
- enclosedline, 2-9
- endline, 2-8
- endpick, 12-2

- endpoint, 2-7
- endpolygon, 2-15
- endqstrip, 2-27
- endselect, 12-8
- ENDTAG, 16-9
- endtmesh, 2-21
- environment variable
 - DISPLAY, 1-2
- environment variables
 - for remote operation, 19-4
- equation
 - depth-cued color, 13-6
 - fog, 13-12
 - size of control point array for NURBS, 14-14
- error
 - floating point, 2-16
 - roundoff, 2-4
- error messages
 - remote, 19-13
- establishing a remote connection, 19-4
- Ethernet, 19-2
- eye coordinate system, 7-3

F

- 4Dwm*, 1-2, 1-3
- fast updates to material, 9-18
- fat polygons, 2-19
- feedback, 17-1 through 17-10
 - data types, 17-2
 - effect of clipping, 17-3
 - IRIS-4D/GT/GTX, 17-4
 - IRIS-4D/VGX, 17-7
 - IRIS Indigo, 17-6
 - Personal IRIS, 17-6
 - sample program, 17-8
- feedback, 17-2
- field of view, 7-5

- finding out if a device is queued, 5-4
- finding the cursor, 5-7
- finding the monitor type, 5-18
- finish, 19-3
- flicker, cause of in animation, 6-2
- floating point calculations, 7-5
- floating point precision, 2-16
- fog, 13-10 through 13-14
 - density, 13-12
 - equation, 13-13
 - modes, 13-14
 - per-pixel, 13-12
- fog characteristics, 13-12
- fogvertex, 13-10, 13-14
- font
 - creating, 3-7 through 3-10
 - selecting, 3-12
- Font Manager Library, 1-4
- fonts, 3-7 through 3-14
- frame buffer, 4-3, 4-17, 6-1
 - standard, 10-2
- frame buffers, 10-1 through 10-7
 - RealityEngine, 10-2
- frame rate, 6-3
- front-facing polygons, 2-24
- front buffer, 6-1
- frontbuffer, 6-2
 - zdraw, 8-10
- frontface, 8-22
- frustum, 7-5, 7-9, 7-10

G

- gamma correction, 4-23
- gamma ramp, 4-24
- gammaramp, 4-23, 4-24
- Gaussian filter for antialiasing, 15-35

- gconfig
 - doublebuffer, 6-2
 - multimap, 4-22
 - onemap, 4-22
 - RGBmode, 4-5
 - singlebuffer, 6-2
- generating a numeric identifier
 - for an object, 16-4
- genobj, 16-4
- gentag, 16-10
- geometric surface, 14-16
 - NURBS, 14-13
- geometry
 - bgn/end structure of, 2-2
 - definition, 2-1
- Geometry Engine, 17-2
- Geometry Pipeline, 1-1
 - feedback, 17-1
 - remote operation considerations, 19-3
- getbackface, 8-22
- getbuffer, 6-7
- getbutton, 5-8
- getcmmode, 4-22
- getcolor, 10-5
- getcpos, 3-4
- getcursor, 5-7, 10-21
- getdcm, 13-4
- getdev, 5-8
- getdisplaymode, 6-7
- getdrawmode, 10-7
- getfont, 3-14
- getgdesc, 1-7
 - GD_BITS, 4-3, 10-2
 - GD_BITS_NORM_SING_ALPHA, 15-7
 - GD_CIFRACT, 4-4
 - GD_DITHER, 4-4
 - GD_FOGVERTEX, 13-11
 - GD_LIGHTING_TWOSIDE, 9-17
 - GD_PIXPERSPECT, 18-33

- GD_POLYMODE, 2-30
- GD_ZMAX, 8-7
- GD_ZMIN, 8-7
- getgpos, 2-37, 7-3
- getlstyle, 2-10
- getlwidth, 2-11
- getmap, 4-22
- getmatrix, 7-29
- getmcolor, 4-18, 10-5
- getmonitor, 5-18
- getnurbsproperty, 14-21
- getopenobj, 16-8
- getothermonitor, 5-19
- getscrbox, 7-29
- getscrmask, 7-28
- getting color information, 4-18
- getting font information, 3-14
- getting the size of the viewport, 7-27
- getvaluator, 5-7
- getvideo, 5-20
- getviewport, 7-27
- getwritemask, 10-10
- getzbuffer, 8-3
- gflush, 2-2, 19-2
- ghost devices, 5-13
- GLC_OLDPOLYGON, 2-31
- GLC_ZRANGEMAP, 13-3
- glcompat, 13-3
- GL Demos*, 14-1
- global state attributes, 1-10
- glresources, 5-2
- GL timers, 5-12
- GLX group of commands, 5-2
- Gouraud shading, 4-8, 4-10, 4-12, 10-7
 - color map mode, 4-17
 - performance, 4-9

- graphical object
 - creating, 16-2
 - defintion, 16-1
 - sample, 16-3
- graphical objects, 16-1
- graphics development environment, 1-1
- Graphics Library, xxi
 - network transparency, 1-2
- Graphics Library,query version, 1-8
- graphics primitives, 2-1
- graphics server, 19-12
- graphics server daemon, 19-12
- gRGBcolor, 4-8
- gRGBmask, 10-10
- gross clipping, 3-4
- gselect, 12-8
- gsync, 6-8

H

- hardware
 - cursor, 10-17
 - feedback, 17-1 through 17-10
 - query, 1-7
- haze, 13-10
- hidden-surface removal, 8-1 through 8-26
- hierarchical drawing, 7-20
- hierarchical object, 16-5
- hierarchies, 16-1
- hierarchy
 - diagram, 16-6
- highlights on surfaces, 9-3
- hint
 - noport, 4-24
 - scrsubdivide, 18-45
 - texbind, 18-45
 - texdef2d, 18-44
 - texture caching, 18-45

- texture performance, 18-43
- hits, 12-1
- hostname, 1-3
- how a writemask works, 10-8

I

- iconsize, 5-14
- immediate mode, 2-1
- implicit representations, 14-2
- improving lighting performance, 9-22
- incident light, 9-2
- include files, 1-2
- independent variable, 14-2
- indirect light, 9-3
- inetd* daemon, 19-13
- infinite light source, 9-9
- infinite viewpoint, 9-10
- initnames, 12-5
- INPUTCHANGE, 5-3, 5-14
- input devices, 5-10 through 5-17
 - classes, 5-10
 - cursor, 5-13
 - keyboard, 5-13
- input event, 5-1
- input focus, 5-2
 - testing for change, 5-14
- instructions
 - how to create a cursor, 10-19
 - how to draw a NURBS surface, 14-14
 - how to draw old-style surface patches, 14-33
 - how to set up depth-cueing, 13-2
 - how to set up fog, 13-11
 - how to set up lighting, 9-4
 - how to set up picking, 12-2
 - how to set up queueing, 5-3
 - how to set up texture mapping, 18-6

- how to use man pages, xxv
- how to use sample programs, xxii
- Internet, 19-12
- interpolation
 - texture, 18-38
- interrupts limited for remote operation, 19-5
- IRIS-4D/GT/GTX feedback, 17-4
- IRIS-4D/VGX
 - clearing of z-buffer, 8-9
 - double-buffered overlay/underlay, 10-7
 - feedback, 17-7
 - fog, 13-10
 - mesh handling, 2-28
 - stenciling, 8-16
 - subpixel, 15-4
 - swapping multiple buffers, 6-3
- IRIS Font Manager, 3-1
- IRIS Indigo
 - allows local lights in color map mode, 9-25
 - backface removal, 8-21
 - dithering, 4-4
 - feedback, 17-6
 - fog, 13-10
 - no overlay/underlay on starter system, 10-1
 - pixel formats, 11-3
 - querying for perspective correction of texture, 18-33
 - texture perspective correction in SW, 18-32
 - z-buffer in software, 8-2
- IRIX, 1-1
- isobj, 16-4
- isqueued, 5-4
- istag, 16-9
- izoom, 18-29

J

- jumps limited for remote operation, 19-5

K

keepaspect, 7-6

kernel

filter, 18-20

keyboard

ringing the bell, 5-17

setting duration of bell, 5-17

turning on/off key click, 5-16

turning on/off lights, 5-16

keyboard device, 5-13

knot multiplicity, 14-8

knots, 14-5, 14-8

knot sequence, 14-8

L

-lc_s, 1-4

-lfm_s, 1-4

-lgl_s, 1-4

-lm_s, 1-4

-lsun, 1-4

labeling dial and button box indicators, 5-17

lampoff, 5-16

lampon, 5-16

L_{COLOR}, 9-9

Level-of-Detail, 18-24, 18-29

library

shared C, 1-4

shared Graphics Library, 1-4

light

ambient, 9-4, 9-9

attenuation, 9-14

emitted, 9-4

incident, 9-2

reflected, 9-2

wavelength, 9-2

lighting, 9-1 through 9-27

binding, 9-11

changing currently bound conditions, 9-13

defaults, 9-13

enabling, 9-11

how calculated, 9-6

in color map mode, 9-24

moving a light from frame to frame, 9-13

performance, 9-22

performance of updates, 9-19

two-sided, 9-18

lighting model, 9-10

light source

targets, 9-11

light sources, 9-9

limitations of lighting in color map mode, 9-24

linear interpolation of texture, 18-38

line of sight, 7-13, 7-14

lines

antialias, 15-15

old-style, 2-39

style, 2-10

width, 2-11

linestyle

definition, 2-10

specifying, 2-10

linewidth, 2-11

linewidthf, 2-11

linking, 1-4

Live Video Digitizer, 5-21

lmbind, 9-12, 9-21

LMC_AD, 9-20

LMC_AMBIENT, 9-20

LMC_COLOR, 9-20

LMC_DIFFUSE, 9-20

LMC_EMISSION, 9-20

LMC_NULL, 9-20

LMC_SPECULAR, 9-20

lmcolor, 9-19

performance, 9-19

- lmdef, 9-7, 9-8, 9-12, 9-14, 9-21
- LMNULL, 9-7
- loadmatrix, 7-29
- loadname, 12-4
- local host, 1-3
- LOCALVIEWER, 9-10
- LOCALVIEWER performance, 9-23
- local viewpoint, 9-10
- LOD, 18-24, 18-29
- logicop, 15-4
- lookat, 7-12, 7-14
- lrectread, 11-6, 11-16, 18-10
- lrectwrite, 11-7
- lRGBrange, 13-4
- lsetdepth, 8-7, 13-3, 13-5
- lshaderange, 13-4, 13-5, 13-10
- lsrepeat, 2-11

M

- makeobj, 16-2
- maketag, 16-9
- makewhatis*, xxvi
- making a light move from frame to frame, 9-13
- man*, xxv
- man pages, xxv
- mapcolor, 4-16, 10-5, 13-5
- mapw, 16-13
- mapw2, 16-13
- mask
 - linestyle, 2-10
 - pattern, 2-16
- material
 - defining, 9-8
 - different front and back, 9-18
 - fast updates, 9-18
 - transparent, 9-21

- matrix mode, 7-19
- matrix multiplication, 7-19
- maximum z value, querying for, 8-7
- MAXLIGHTS light sources limit, 9-11
- memory management
 - objects, 16-12
 - texture, 18-45
- meshes 2-21 through 2-28
 - changing vertex sequence, 2-23
 - drawing, 2-21
 - vertex sequence, 2-22
- minification filters, 18-17
- minimum z value, querying for, 8-7
- MIPmap
 - filter kernel, 18-20
 - quadlinear, 18-18
 - trilinear, 18-18
- mixed-model programming, 5-2
- mmode, 7-20
 - for fog, 13-11
 - MTEXTURE, 18-33
 - pushmatrix, 7-21
- mode
 - color map, 4-15 through 4-20
 - double buffer, 6-1, 6-2
 - fog, 13-14
 - for bitplanes, 10-5
 - for normal frame buffer, 10-6
 - multimap, 4-22
 - onemap, 4-22
 - querying color, 4-8
 - querying for display, 6-7
 - RGB, 4-5 through 4-15
 - setting up for animation, 6-2
 - single buffer, 6-1, 6-2
- modeling transformations, 7-16 through 7-19
- ModelView matrix, 7-3, 7-19, 7-29, 9-11, 9-13, 9-22

- monitor
 - color display, 4-2
 - other types, 5-19
 - querying for type, 5-18
 - setting type, 5-18
- Motif, 5-2
- move, 2-39
- MPROJECTION, 7-19
- msalpha, 15-47
- mspattern, 15-45
- mssize, 15-38
- mswapbuffers, 6-3
- MTEXTURE, 7-19
- multimap mode, 4-22
- multimatrix mode in lighting, 9-11
- multiple server I/O, 5-2
- multisampling, 15-37 through 15-47
- multmatrix, 7-29
- MVIEWING, 7-19

N

- name stack, 12-2, 12-4
- naming the output file, 1-4
- network-transparent feature of GL, 1-2
- network-transparent GL, 19-1 through 19-16
 - callback limit, 19-5
 - configuration, 19-12
 - connection, 19-4
 - DISPLAY, 1-2
 - error messages, 19-13
 - limitations, 19-5
 - linking, 19-2
 - linking for NIS, 1-4
 - non-IRIS hosts product, 19-2
 - sample application, 19-1
 - server, 1-3
- Network Information Service, 1-4, 19-12

- network transparency, 19-1
- newtag, 16-9
- NIS, 1-4, 19-12
- noise, 5-7, 5-12
- non-IRIS hosts, 19-2
- non-recommended subroutines, 2-37 through 2-42
- Non-uniform Rational B-splines, 14-1
- non-uniform scaling, 7-18
- noport, 4-24
- normal, current, 9-5
- NORMALDRAW, 10-6
 - swapping buffers in, 6-4
 - writemasks, 10-9
- normal frame buffer modes, 10-6
- normalized coordinate system, 7-3
- normals, 9-5
 - non-unit-length, 9-6
- NTSC, 5-21
- NURBS, 14-1 through 14-27
 - circle, 14-12
 - color surface, 14-17
 - conic sections, 14-12
 - cross-plot, 14-4
 - culling, 14-21
 - curve as a trimming curve, 14-20
 - curves, 14-11 through 14-13
 - definition, 14-11
 - enabling error messages, 14-21
 - formula for size of control point array, 14-14
 - further reading, 14-27
 - geometric surface, 14-16
 - offset in control point array, 14-15
 - piecewise linear trim curve, 14-20
 - properties, 14-21 through 14-22
 - surfaces, 14-13 through 14-18
 - tessellation, 14-21
 - texture surface, 14-17
 - trimming, 14-18 through 14-21
 - weight, 14-10

nurbscurve, 14-11
nurbssurface, 14-13

O

-o, 1-4
objdelete, 16-10
object
 closing a definition, 16-2
 creating, 16-2
 deleting, 16-4
 drawing, 16-4
 editing, 16-8
 hierarchy, 16-5
 identifying by number, 16-4
 memory management, 16-12
 tags, 16-9
 testing for existence, 16-4
object coordinate system, 7-2
objinsert, 16-10
objreplace, 16-11
offset, 14-15
old-style
 curves, 14-28, 14-28 through 14-33
 drawing, 2-30 through 2-42
 lines, 2-39
 pixels, 11-16
 points, 2-38
 polygons, 2-40
 polygon subroutines, 2-41
 surfaces, 14-28, 14-33 through 14-34
on-line man pages, xxv
onemap mode, 4-22
opening a window, 1-6
operating system, 1-1
order of pixel operations, 11-16
order of the polynomial, 14-5
order of transformations, 7-18, 7-19
order of vertices in lighting, 9-5

origin
 cursor, 10-20
ortho, 7-10, 7-11
ortho2, 2-4, 7-11, 12-3
orthographic projection, 7-4, 7-10
output file
 naming, 1-4
OVERDRAW, 10-6
 swapping buffers in, 6-4
 writemasks, 10-9
overlapping polygons, 8-2
overlay
 bitplanes, 10-3
 definition, 10-1
 non-recommended routines, 10-7
 user-defined, 10-15
overlay, 10-15

P

packing pixel data, 11-10, 11-14
PAL, 5-21
parameter, 14-2
parameterization, in textures, 18-33
parametric bicubic surface, 14-33
parametric representations, 14-2
parametric surface, 14-13
patterns, 2-16
pclos, 2-40
pclose warning, 2-40
per-pixel fog, 13-12
performance
 animation, 6-6
 font, 3-1
 Gouraud shading, 4-9
 improving lighting, 9-22
 lmc color limitations, 9-19
 optimizing polygon, 2-12

- texture, 18-43 through 18-46
 - texture alpha, 18-45
 - two-sided lighting, 9-23
 - permission for connections, 19-4
 - Personal IRIS
 - 8-bit pixel formats, 11-3
 - dithering, 4-4
 - feedback, 17-6
 - no overlay/underlay on B-bit models, 10-1
 - set Gouraud or depthcue for dithering, 4-4
 - speeding up `czclear`, 8-9
 - perspective, 7-5
 - perspective projection, 7-4
 - phosphors, 4-2
 - pick, 12-2
 - picking, 12-1 through 12-8
 - size of region, 12-4
 - picksize, 12-4
 - piecewise linear trimming curve, 14-20
 - piecewise polynomial, 14-5
 - pixel, 4-2, 4-4, 8-23
 - centering, 15-4
 - pixels
 - accessing, 11-2
 - centering, 11-4
 - centering on integers, 7-11
 - changing fill directions, 11-12
 - copying, 11-7
 - customizing operations, 11-10
 - formats, 11-2
 - formats of IRIS Indigo, 11-3
 - high-performance reading/writing, 11-3
 - old-style subroutines, 11-16
 - order of operations, 11-16
 - packing/unpacking, 11-14
 - setting read source, 11-4
 - writing, 11-7
 - `pixmode`, 11-10
 - `PM_EXPAND`, 11-11
 - `PM_OFFSET`, 11-16
 - `PM_SHIFT`, 11-10
 - `PM_SIZE`, 11-14
 - `PM_STRIDE`, 11-13
 - `pmv`, 2-40
 - `pntsize`, 2-7
 - `pntsizef`, 2-7
 - `pntsmooth`, 15-15
 - color map mode, 15-11
 - point-sampled polygons, 2-17 through 2-20
 - point and click interface, 12-1
 - point light source, 9-9
 - points
 - antialiased color map, 15-11
 - drawing, 2-7
 - old-style, 2-38
 - size, 2-7
 - polarview, 7-12, 7-13
 - `polf`, 2-41
 - polling, 5-2, 5-7 through 5-10
 - button, 5-8
 - multiple valuator, 5-8
 - valuator, 5-7
 - `poly`, 2-41
 - polygon
 - antialiased, 15-22
 - bowtie, 2-15
 - concave, 2-13
 - convex, 2-12
 - definition, 2-12
 - display anomalies, 2-20
 - distortion, 2-16
 - drawing, 2-15
 - fat, 2-19
 - fill patterns, 2-16
 - front-facing, 2-24
 - Gouraud shaded, 4-8
 - non-planar, 2-16
 - old-style, 2-40
 - optimizing performance, 2-12

- outlined in old-style drawing, 2-31
- overlapping, 8-2
- point-sampled, 2-17 through 2-20
- rendering, 2-29
- simple, 2-12
- true area of, 2-19
- polyline
 - definition, 2-8
 - drawing, 2-8
- polymode, 2-29
- polynomials, 14-5
- pop-up menu callbacks
 - limited for remote operation, 19-5
- pop-up menus, 10-3
- popmatrix, 7-21
- popname, 12-5
- popviewport, 7-28
- POSITION, 9-9, 9-17
- position of a light source, 9-10
- precision in hardware calculations, 2-16
- primitive
 - character, 3-2
 - lighted, 9-4
 - line, 2-8
 - NURBS, 14-11
 - overlapping, 15-13
 - point, 2-7
 - polygon, 2-12
 - q-strip, 2-21, 2-27 through 2-28
 - smooth, 15-10
 - t-mesh, 2-21 through 2-27
- primitives, 2-1, 2-6
- program relinking to run remotely, 19-2
- projection
 - orthographic, 7-4, 7-10
 - perspective, 7-4
- Projection matrix, 7-5, 7-19, 7-30
- projection transformations, 7-4 through 7-12
 - for picking, 12-3
 - window, 7-9

- properties of NURBS, 14-21
- pruning, 16-5
- PUPDRAW, 10-6
- pushattributes/popattributes
 - when calling objects, 16-5
- pushmatrix, 7-21
- pushname, 12-4
- pushviewport, 7-28
- pwlcurve, 14-20
- PYM_FILL, 2-29
- PYM_HOLLOW, 2-30
- PYM_LINE, 2-29
- PYM_POINT, 2-29

Q

- q-strips, 2-27 through 2-28
 - definition, 2-21
 - reducing to triangles, 2-28
- qdevice, 5-4, 5-13
- qenter, 5-4
- QFULL, 5-3
- qgetfd, 5-5
- qread
 - reading the queue, 5-5
 - remote, 19-5
- qreset, 5-3, 5-5
- qtest, 5-5
- quadlinear MIPmap, 18-18
- queuing, finding out if a device is queued, 5-4
- query
 - alpha bitplanes, 15-7
 - current linestyle, 2-10
 - depth-cueing, 13-4
 - font information, 3-14
 - system version, 1-7

- queue, 5-2
 - creating entries, 5-4
 - emptying, 5-3
 - limit, 5-3
 - reading, 5-5
 - reading multiple entries, 5-5
 - resetting, 5-5
 - testing for event, 5-5
- queueing, 5-2, 5-3 through 5-7
 - disabling, 5-4
 - enabling, 5-4
 - how to set up, 5-3
 - keyboard, 5-13
 - maximum efficiency, 5-3
 - window manager events, 5-14
- quitting a window, 5-14

R

- Raster Manager (RM) boards, 15-37
- rational, 14-11
- readdisplay, 11-5
- readfds, 5-6
- reading pixels, 11-4
- reading location of, 5-6
- reading man pages, xxv
- reading multiple queue entries, 5-5
- reading video registers, 5-20
- README for sample programs, xxii
- readsource, 11-4
- RealityEngine
 - antialiasing, 15-36 through 15-47
 - bitplanes, 10-16
 - color, 4-5
 - color map size, 4-7
 - feedback, 17-8
 - fog, 13-10
 - framebuffers, 10-2
 - multisample buffer, 10-2

- stenciling, 8-16
- stereo buffering, 10-4
- swapbuffers, 6-3
- z-buffer size, 8-2
- recording events, 5-12
- rectangle subroutines, 2-32
- rectangles, drawing, 2-32
- rectcopy, 11-7
- rectread, 11-5
- rect/rectf, 2-32
- rectwrite, 11-7
- rectzoom, 11-7
- REDRAW, 5-3, 5-14
- REDRAWICONIC, 5-14
- reducing graphics jerkiness
 - in remote applications, 19-3
- reflectance, 9-3
 - diffuse, 9-3
 - specular, 9-3
- reflected light, 9-2
- reflection mapping, 18-34
- remote
 - configuration, 19-12
 - connection, 19-4
 - error messages, 19-13
 - interrupts, 19-5
 - jumps, 19-5
- remote display of graphics, 19-1
- REMOTEHOST, 19-4
- removing backfacing polygons, 8-22
- removing frontfacing polygons, 8-22
- rendering
 - averaging with accumulation, 15-31
 - bowtie polygons, 2-16
 - concave polygons, 2-13
 - improving polygon performance, 2-31
 - into 2 buffers, 6-3
 - NURBS, 14-21
 - polygons, 2-29

- resetting the queue, 5-5
- resolution of the z-buffer, 8-3
- RGB mode, 4-5 through 4-15
 - antialiasing, 15-13
 - depth-cueing, 13-6
 - double buffering in, 6-2
 - writemasks, 10-9
- RGBmode, 4-5
- RGBsize, 15-41
- RGB triple, 4-2, 4-5, 4-7
- RGBwritemask, 10-9
- right-hand rule, 2-24, 9-5
- ringbell, 5-17
- ringing the keyboard bell, 5-17
- rlogin, 19-4
- rot, 7-17
- rotate, 7-17
- round-off errors, 11-4
- roundoff error, 2-4

S

- sample program
 - aliasing, 15-2
 - accumulation, 15-32
 - color map line, 15-17
 - points in color map mode, 15-11
 - RGB mode line, 15-19
 - RGB mode points, 15-13
 - RGB polygon, 15-22
 - RGB polygon corrected, 15-27
 - RGB polygon patch, 15-24
 - blending, 15-9
 - cursors, 10-21
 - depth-cueing, 13-7
 - drawing a hexagon, 2-15
 - drawing a polygon, 2-9
 - drawing lines, 2-5, 2-8
 - drawing meshes, 2-25

- drawing old-style rectangles, 2-33
- drawing points, 2-7
- drawing with writemasks, 10-12
- feedback, 17-8
- NURBS, 14-22
- object editing, 16-11
- picking, 12-5
- selecting, 12-9
 - vertex subroutines, 2-4
 - writemasks, 10-10
- sample programs
 - where located, xxii
- scale, 7-18
- scaling of z coordinates, 8-7
- sclear, 8-19
- scrbox, 7-28
- screen
 - setting blanking time, 5-18
 - turning on/off, 5-18
- screen boxes, 2-33
- screenmask, 3-3, 7-27
- scrmask, 7-27, 11-7
- scrsubdivide, 18-33, 18-38 through 18-40
- searching man pages by key word, xxv
- select, 5-5
- selecting, 12-8 through 12-10
- selecting a bitplane for drawing, 10-1
- selecting region, 12-8
- server, 1-3
- servers, 19-1
- setbell, 5-17
- setcursor, 10-21
- setdblights, 5-17
- setlinestyle, 2-10
- setmap, 4-22
- setmonitor, 5-18
- setnurbsproperty, 14-21
- setting screen blanking time, 5-18

- setting the duration of keyboard bell, 5-17
- setting the monitor type, 5-18
- setting up lighting, 9-4
- setvaluator, 5-16
- setvideo, 5-19
- shademodel, 4-9
- shademodel(FLAT), 4-9
- shademodel(GOURAUD), 4-9
- shared Graphics Library, 1-4
- shared library, 1-2
 - C, 1-4
 - for network-transparent GL, 1-2
- Sharpen, 18-23 through 18-27
- shifting pixel data, 11-10
- SHININESS, 9-8
- simple polygon, 2-12
- simultaneous clearing
 - of color and z-buffer, 8-8
- singlebuffer, 6-2
- single buffer mode, 6-1, 6-2
- sizing the viewport, 7-26
- SkyWriter
 - clearing of z-buffer, 8-9
 - double-buffered
 - overlay/underlay, 10-7
 - fog, 13-10
 - mesh handling, 2-28
 - per-pixel fog, 13-12
 - perspective correction
 - of texture in hardware, 18-33
 - stenciling, 8-16
 - subpixel, 15-4
 - swapping multiple buffers, 6-3
 - texture perspective correction, 18-32
 - TX_MIPMAP_TRILINEAR filter, 18-19
- sleep*, 2-6
- smoke, 13-10
- Spaceball device, 5-15
- SPECULAR, 9-8
- specular reflectance, 9-3
 - defining for material, 9-8
 - depends on viewpoint, 9-8
- spline, 14-5
- SPOTDIRECTION, 9-16
- SPOTLIGHT, 9-16
- spotlight, 9-16
 - attenuation, 9-17
 - direction, 9-16
- sprintf*, 4-7
- SRC_AUTO, 11-4
- SRC_BACK, 11-4
- SRC_FRAMEGRABBER, 11-4
- SRC_ZBUFFER, 11-4
- SS_DEPTH, 18-39
- stack matrix, 7-20
- STARTTAG, 16-9
- stencil, 8-16
- stenciling, 8-16 through 8-21
- stencil planes, 8-18
 - clearing, 8-19
 - enabling, 8-19
- stensize, 15-41
- stensize, 8-18
- stereo buffering, 10-4
- stowed windows, 5-14
- structure, using offsets to specify data in, 14-15
- strwidth, 3-5
- subimage, 11-13, 18-29
- subpixel, 15-4, 15-10
- subpixel accuracy, 15-4 through 15-5
- surface, transparent, 9-21
- surface normals, 9-5
- surface patch, 14-33
- surfaces, old-style, 14-28, 14-33 through 14-34
- swapbuffers, 6-3
- swapinterval, 6-7

- swapping buffers, defining interval for, 6-7
- swapping multiple buffers, 6-3
- swaptmesh, 2-23, 2-24
- swimming effect of texture, 18-37
- swritemask, 8-19

T

- 3-D textures, 18-4
- t-mesh, 2-21 through 2-27
 - definition, 2-21
- t2d, 18-33
- t2f, 18-33
- t2i, 18-33
- t2s, 18-33
- tags, 16-9
- TCP socket connections, 19-12
- tessellation
 - controlling the fineness of for NURBS, 14-21
 - definition, 14-21
- testing a window
 - for closure, 5-14
 - for quitting, 5-14
 - for stowing, 5-14
 - icon needed, 5-14
 - redraw needed, 5-14
- testing for an existing object, 16-4
- testing for button input, 5-6
- testing for change of input focus, 5-14
- testing the queue for events, 5-5
- tevdef, 18-40
- texbind, 18-11
- texdef2d, 18-11
- texdef3d, 18-11
- texels, 18-3
- texel size, 18-13
- texgen, 18-33

- text, drawing, 3-2
- texture, 18-1 through 18-50
 - alpha hints, 18-45
 - automatic generation, 18-33 through 18-34
 - blending, 18-41
 - blurring, 18-21
 - components, 18-7
 - contour, 18-35
 - coordinates, 18-33 through 18-35
 - decals, 18-42
 - environment, 18-40 through 18-42
 - filters, 18-16 through 18-21
 - format of image array, 18-9
 - image, 18-6
 - interpolation, 18-38
 - look-up tables, 18-36
 - magnification, 18-16
 - memory management, 18-45
 - minification, 18-16, 18-17
 - MipMap filter, 18-17 through 18-21
 - MipMap performance, 18-44
 - NURBS surface, 14-13, 14-17
 - performance, 18-43 through 18-46
 - perspective correction, 18-32
 - reflection mapping, 18-34
 - repeating, 18-12
 - scrsubdivide hints, 18-45
 - setting up, 18-6
 - swimming, 18-37 through 18-39
- texture mapping
 - definition, 18-3
- TG_CONTOUR, 18-34
- TG_LINEAR, 18-34
- TG_SPHEREMAP, 18-34
- tie, 5-6
- timer devices, 5-12
- tokens, 19-2
 - window manager, 5-14
- transformation, user-defined, 7-29
- transformation matrices, 7-19
- transformation matrix, 7-16, 7-19

- transformations, 7-2
 - order of, 7-18, 7-19
- translate, 7-17
- transparency, 8-23, 9-21
 - blending, 15-5
- transparent surfaces, 9-21
- tree, 16-6
- trilinear MIPmap, 18-18
- trimming a NURBS surface,
 - 14-18 through 14-21
- trimming curves, 14-18
 - orientation, 14-18
 - piecewise linear, 14-20
- trimming loop, 14-18
- turning on/off
 - cursor, 5-6
 - depth-cueing, 13-4
 - dial and button box lights, 5-17
 - keyboard click, 5-16
 - keyboard lights, 5-16
 - screen, 5-18
- TV_BLEND, 18-40, 18-41
- TV_DECAL, 18-41, 18-42
- TV_MODULATE, 18-40, 18-41
- twist, 7-13, 7-14
- two-sided lighting, 9-17
 - performance, 9-23
- TWOSIDE, 9-17
- TX_MAGFILTER, 18-12
- TX_MINFILTER, 18-12
- TX_REPEAT, 18-12
- TX_TILE, 18-13
- TX_WRAP, 18-12

U

- UNDERDRAW, 10-6
 - swapping buffers in, 6-4
 - writemasks, 10-9
- underlay
 - bitplanes, 10-3
 - definition, 10-1
 - non-recommended routines, 10-7
 - user-defined, 10-15
- underlay, 10-15
- unpacking pixel data, 11-10, 11-14
- unqdevice, 5-4
- updating active materials, 9-18
- user-defined transformation, 7-29
- user input in a GL program, 5-1
- user interface, 5-1

V

- valuator
 - names of, 5-12
 - noisy, 5-7
 - polling, 5-7
 - polling multiples, 5-8
- version, querying, 1-7
- vertex, 2-1
 - normals, 2-24
 - order for front-facing polygons, 2-24
 - subroutines, 2-2, 2-3
- vertex ordering, importance of in lighting, 9-5
- video, 5-17 through 5-21
 - enabling the LVD option, 5-21
 - formats, 5-21
 - reading registers, 5-20
 - setting hardware registers, 5-19
- videocmd, 5-21
- viewing transformation, 7-3
- viewing transformations, 7-12 through 7-16

- viewpoint
 - defining, 7-13
 - for lighting, 9-10
- viewport, 3-3, 7-26 through 7-29
- viewport, 7-26
- volume rendering, 18-5

W

- wavelength of light, 9-2
- weight of control points, 14-10
- window
 - opening, 1-6
 - quitting, 5-14
 - testing for closure, 5-14
 - testing for iconification, 5-14
 - testing if redraw needed, 5-14
- window, 7-9
- window coordinate system, 7-3
- window manager
 - 4Dwm*, 1-2, 1-3
 - queueing events from, 5-14
- window manager tokens, 5-14
- WINFREEZE, 5-14
- WINQUIT, 5-14
- WINSHUT, 5-14
- WINTHAW, 5-14
- wireframe, definition, 2-1
- wmpack, 10-10
- world coordinate system, 7-3
- writemask, 10-7
 - definition, 10-7
 - how it works, 10-8
- writemask, 10-9, 11-7
- writemasks, 10-7 through 10-15
- writing pixels, 11-7
- writing to the z-buffer, 8-10

X

- X client, 1-2
- X input model, 5-2
- xman*, 1-2
- X server, 1-2, 1-3, 5-2
- xterm*, 1-2
- Xt routines, 5-2
- X Window System, 1-2, 5-2

Z

- z-buffer, 8-2 through 8-15
 - changing comparison test, 8-14
 - clearing, 8-8
 - enhancing resolution of, 8-3
 - writing directly to, 8-10
- zbuffer, 8-3
- z coordinates, scaling, 8-7
- zdraw, 8-10
- zfar, 8-3
- zfar*, 13-5
- znear, 8-3
- znear*, 13-5
- z values for depth-cueing, 13-3
- zwritemask, 8-15
- z-buffer
 - zwritemask, 8-15

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-1702-020.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
 - On the Internet: techpubs@sgi.com
 - For UUCP mail (through any backbone site): *[your_site]!sgi!techpubs*
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

Technical Publications
Silicon Graphics, Inc.
2011 North Shoreline Boulevard, M/S 535
Mountain View, California 94043-1389