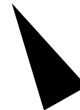
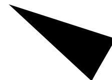
XGL™ Programmer's Guide





SunSoft, Inc.
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

THE NETWORK IS THE COMPUTER™

Copyright 1997 Sun Microsystems, Inc. 2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A. All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

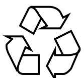
Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, SunSoft, SunDocs, SunExpress, XGL, and Solaris are trademarks, registered trademarks, or service marks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK and Sun™ Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-3(a).

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.



Please
Recycle

Copyright 1997 Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, Californie 94043-1100 Etats-Unis. Tous droits réservés.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Des parties de ce produit pourront être dérivées des systèmes Berkeley BSD licenciés par l'Université de Californie. UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

Sun, Sun Microsystems, le logo Sun, SunSoft, SunDocs, SunExpress,XGL, et Solaris sont des marques de fabrique ou des marques déposées, ou marques de service, de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

L'interface d'utilisation graphique OPEN LOOK et Sun™ a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui en outre se conforment aux licences écrites de Sun.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" ET AUCUNE GARANTIE, EXPRESSE OU IMPLICITE, N'EST ACCORDEE, Y COMPRIS DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DE LA PUBLICATION A REPONDRE A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ELLE NE SOIT PAS CONTREFAISANTE DE PRODUIT DE TIERS. CE DENI DE GARANTIE NE S'APPLIQUERAIT PAS, DANS LA MESURE OU IL SERAIT TENU JURIDIQUEMENT NUL ET NON AVENU.

Contents

Preface. xxxi

New Features xxxvii

1. Introduction to XGL 1

Overview of XGL Functionality 1

Direct Graphics Access 4

Introduction to XGL Objects and Graphical Organization 6

System State Object and Generic Operators 7

Device Object 8

2D and 3D Context Objects. 9

Color Map Object. 10

Transform Object 10

Line Pattern Object 11

Light Object 12

Stroke Font Object 13

Marker Object. 13

Pcache Object 13
Gcache Object. 14
Texture Map Object and MipMap Texture Object 14
2. Getting Started with XGL Programming 15
Compiling and Running XGL Programs. 15
Run-time Considerations 17
XGL Example Programs 18
If You Experience Poor Performance for Local Rendering . 18
Rendering Remotely Through PEX 19
Basic XGL Concepts. 20
XGL and the X Window System Environment. 20
How an XGL Application Works 23
XGL Drawing Primitives 24
Handling 2D and 3D Data 25
More on XGL Object-Based Programming 27
Example Program. 30
Creating the Window and Registering Callback Procedures 34
Opening XGL 34
Creating a Window Raster Device Object. 34
Creating a Context Object. 35
Rendering Geometry. 35
Objects Provided at XGL Initialization 36
Programming Tips 37
General Tips 37

XView Tips 38
OLIT Tips 39
XGL Limitations. 40
3. System State Information and Generic Operators 43
Introduction to the System State Object 43
System State Operators 43
System State Attributes 44
Error Detection and Reporting. 45
Error Notification Function 46
Error Types and Categories 47
Generic XGL Operators. 49
4. Devices 53
Introduction to Device Objects. 53
Creating Device Objects 54
Window Raster Device Object 55
Stream Device Object. 57
Raster Object Attributes 58
General Raster Attributes 58
Window Raster Attributes 61
Memory Raster Attributes 64
CGM Device Object 66
Creating a CGM Device 66
Creating a Picture 68
CGM Metafile Device Attributes 69

XGL CGM Line Patterns 71
XGL CGM Markers 72
Important Notes on Integrating XGL with a Windowing System 73
Mixing XGL and Xlib Drawing Routines 73
Window Resize Events 74
Device Example Programs 74
XGL and Xlib 75
XGL and the XView Toolkit 79
XGL and the OLIT Toolkit 83
Determining Device Acceleration 86
Inquire Example Program. 88
Transparent Overlay Windows. 94
Performance Cases for Transparent Overlays 94
Creating an Overlay Window. 96
Rendering to an Overlay Window Using the XGL API . . . 98
Clearing the Overlay Window Using the XGL API. 99
Transparent Overlay Example Program 99
5. Color 115
Introduction to XGL Color 115
Indexed Color Space 116
RGB Color Space 116
XGL Color Pipeline 116
Introduction to the Color Map Object 118
Creating a Color Map Object 118

Color Tables 119
Color Table Attributes 119
Creating a Color Table 120
Simple Color Table Example 121
Color Table Ramps 124
Creating Color Table Ramps 125
Color Ramp Example 126
Color Mapping 130
Color Cubes 130
Dithering 131
Sharing the Window System Color Map 132
Sharing the Window System Color Map Example 134
Color Map Double Buffering 144
Color Map Double Buffering Example 145
XGL and the Kodak Color Management System 150
XGL and KCMS Example Code 151
6. Contexts 159
Introduction to the XGL Context Object 159
Creating a Context Object 160
Context Object Operators 162
Context Stacks 164
Environment Context Attributes 164
Context Attributes for Picking 166
Context Attributes Associated with the View Model 167

Context Attributes Associated with Lighting 167

7. Primitives and Graphics Context Attributes 169

Introduction to XGL Drawing Primitives 169

Description of Drawing Primitives 173

Drawing Primitive Data Structures 177

Point Types 177

Point Lists 177

Facet Structures 178

Bounding Boxes 179

Annotated Primitives 184

Graphics Context Attributes. 185

General Rendering Attributes 186

Context Accumulation Buffer Attributes 187

Hidden Line/ Hidden Surface Removal (HLHSR) 188

Context Line Attributes. 189

Context Marker Attributes 190

Context Curve Attributes 191

Context Surface Attributes 191

Context Stroke Text Attributes 198

Context Annotation Text Attributes 199

Context View and Transform Attributes. 200

Context Paint Attributes 201

Example Programs using XGL Primitives 204

Polygon Example Program. 204

Rectangle Example Program 206
Circle Example Program 209
Raster Primitives 213
Getting and Setting Pixel Values 213
Copying Buffer Contents 213
Raster Primitive Example Programs 215
Example Program for Getting and Setting Pixels 215
Example Program for Copying Pixels 217
Accumulation Buffer 222
Accumulation Operator 223
Accumulation Attributes 225
Accumulation Example 225
8. Rendering NURBS Curves and Surfaces with XGL 229
Introduction to NURBS Curves and Surfaces. 229
NURBS Curves 230
Mathematical Description of a NURBS Curve 230
Rendering a NURBS Curve 233
NURBS Curve Attributes 234
Color Splines 237
Curve Example Programs 237
Bezier Curve Example Program. 237
Circle Curve Example Program 242
NURBS Surfaces 247
Mathematical Description of a NURBS Surface 247

Rendering a NURBS Surface 249
Surface Trimming. 250
Performance Hints for Simple Geometry 253
NURBS Surface Attributes 254
Color Surfaces 258
Displaying Silhouettes on NURBS Surfaces 259
Surface Example Program. 259
Using Geometry Cache for Curve and Surface Rendering 264
Getting the Best Out of XGL NURBS. 265
Further Reading 266

9. Transforms 267

Introduction to the Transform Object 267
Creating a Transform Object. 268
Transform Operators 268
Rotation Operator 269
Scaling Operator 270
Translation Operator 270
Reading and Writing Transforms. 271
Other Transform Operators 273
Transform Attributes 276
Examples Using the Transform Object 276
2D Transform Examples 276
3D Transform Example 286

10. View Model 291

Introduction to the XGL Viewing Pipeline 291
Coordinate Systems 292
How to Use Transforms with the View Model 296
Using XGL 2D and 3D Viewing 297
Model Coordinate Space and Model Transforms 297
World Coordinate Space and the View Transform 298
Virtual Device Coordinate Space and the VDC Transform. 299
MC-to-DC Transform 302
Model Clipping in 3D 303
View Clipping 304
View Example Program. 305
Usage Note 306
11. Text 313
Overview of Text in XGL. 313
Stroke Font Object 314
Stroke Fonts Provided by XGL. 314
Creating a Stroke Font Object. 315
Rendering Stroke Text 316
Stroke Font Object Attributes. 320
Context Stroke Text Attributes. 321
Context Annotation Text Attributes 328
Determining Text Extent 329
Text Encoding Schemes. 331
Stroke Font Example Program 334

Raster Text. 339
Caching Font Information. 340
Rendering Raster Text 341
Raster Text Example Program 342
12. Line Patterns 355
Introduction to the Line Pattern Object. 355
Using the Predefined Line Patterns 356
Creating a Line Pattern Object 357
Defining a New Line Pattern 358
Line Pattern Attributes 359
Context Line Pattern Attributes 361
Line Rendering with a Line Pattern 361
Setting Line Color 362
Edge Rendering with a Line Pattern 363
Setting Edge Color 364
Line Pattern Examples. 364
Patterned Line Example 364
Line Pattern Polygon Edge Example. 368
13. Markers. 373
Introduction to the Marker Object 373
Using the Predefined Markers 374
Creating a New Marker. 375
Context Marker Attributes 376
Marker Example Programs 377

Predefined Marker Example Program. 377
User-defined Marker Example Program 380
14. Picking 385
Introduction to XGL Picking 385
Picking Attributes 386
XGL Picking Operators 388
Clearing the Pick Buffer 388
Identifying Picked Primitives. 388
Picking Example. 389
15. Lighting, Shading, and Depth Cueing 395
Introduction to the 3D Rendering Pipeline. 395
Lighting. 395
Shading 396
Depth Cueing 396
Surface Primitives 397
The 3D Context in Lighting 397
Basic Lighting and Shading Concepts. 397
Colors 397
Reflection Components 398
Light Sources 399
Illumination and Shading. 399
Lighting Equations 400
RGB Lighting 403
Indexed Lighting 404

XGL Lighting Attributes 406
Creating a Light Object 407
Light Operators and Attributes 409
Light Operator 409
Light Attributes 409
3D Context Attributes for Lighting 410
Depth Cueing 411
Applications 411
Linear Depth Cueing. 412
Scaled Depth Cueing. 412
Depth Cueing Attributes. 413
Lighting Examples. 415
Light Facet Example 415
Light Vertex Example 420
Linear Depth Cueing Example. 422
Scaled Depth Cueing Example. 425
16. Caching Geometry 429
Introduction to Geometry Caching in XGL 429
Pcache Object 430
Using Pcache Objects With Immediate Mode Applications 431
Pcache Performance Considerations 432
Creating and Rendering to a Pcache 432
Displaying a Pcache. 434
Gcache Object 436

When To Use a Gcache 436
Creating a Gcache 437
Rendering to a Gcache 437
Gcache Attributes 440
Displaying the Gcache. 443
Gcache Example Programs 444
17. Texture Mapping 449
Overview of Texture Mapping 449
MIP Maps 450
Multiple Textures 450
How Texture Mapping Works in XGL. 451
Overview Steps for Mapping a Texture to a Surface 451
Defining the Texture Data 452
Letting XGL Build the Mip Map 452
Using an Application-Supplied MIP Map 457
Determining the Size of the Base Texture Image 458
Specifying the Properties of the Texture 458
Creating a Texture Map Object. 459
Specifying Texture Properties With the Texture Map Object 459
Associating the Texture Map Object With a 3D Context. . . 467
Additional Texture Map Functionality 468
Creating a Data Map Texture Object 474
Specifying Texture Properties With the Data Map Texture Object 474

Associating the Data Map Texture Object With a 3D Context 477
Additional Data Map Texture Attributes 477
Mapping the Texture to a Surface Primitive 479
Using Data Point Types for Texture Mapping 479
Texture Mapping Example Program 482
A. Changes to the XGL 3.2 Library 493
New Operators 493
Changed Attributes 494
Changed Data Structures 494
B. Changes to the XGL Library From XGL 3.0 through XGL 3.1 495
Changes to the XGL Library at XGL 3.1 495
Changed Operators 495
Deleted Operators 496
New Attributes. 496
Changed Attributes 497
Deleted Attributes 498
New Data Structures 498
Changed Data Structures 499
Changes From XGL 3.0 Through XGL 3.0.2 500
Using the xgl_changes Script 500
Programming Notes for XGL 3.x 500
New Operators. 502
Changed Operators 503
Deleted Operators 505

New Attributes. 506
Changed Attributes 509
Renamed Attributes. 511
Deleted Attributes 513
New Data Structures 514
Changed Data Structures 517
Deleted Data Structures 520

C. Software Rendering Characteristics 521
Antialiasing. 521
Transparency 523

D. The Utility and Main Example Programs 525
ex_utils.c 525
color_main.c 537
copy_raster_main.c 539
dcue_main.c 539
gcache_main.c 541
gspixel_main.c 542
light_main.c 543
lpat_main.c 545
nurbs_main.c 546
pick_2d_main.c 547
prims_2d_main.c 548
tran_main.c. 549
view_main.c 551

E. XGL Errors 559

Device-Independent Nonrecoverable Errors 559

Device-Independent Recoverable Errors 561

GX Frame Buffer Errors. 565

Xlib/PEXlib Errors. 566

A Glossary of XGL Terms. 567

Figures

Figure 1-1 XGL API and Foundation Library. 4
Figure 1-2 XGL in the OpenWindows Environment. 5
Figure 1-3 XGL Class Hierarchy. 6
Figure 1-4 System State and Generic Operators 7
Figure 1-5 Overview of the Device Object 8
Figure 1-6 Context Object Operators and Attributes 9
Figure 1-7 Overview of the XGL Color Models. 10
Figure 1-8 Transformed Geometry 11
Figure 1-9 Line Pattern Example 11
Figure 1-10 Overview of Light Sources. 12
Figure 1-11 Marker Examples. 13
Figure 1-12 Texture Mapping of a MipMap to a Polygon 14
Figure 2-1 High-level View of an XGL Application Program 21
Figure 2-2 Using DGA to Render Locally. 22
Figure 2-3 XGL and Remote Rendering 23
Figure 2-4 Instantiated Objects. 28

Figure 2-5 Output of hello_world.c 31

Figure 4-1 Output of ow_olit.c 75

Figure 4-2 Output of inq.c 88

Figure 5-1 XGL Color Pipeline 117

Figure 5-2 RGB Color Cube.................................... 131

Figure 7-1 XGL Drawing Primitives Illustrated................ 172

Figure 7-2 Context Attributes Pipeline 202

Figure 7-3 Output of prims_2d_pgon.c 204

Figure 7-4 Output of prims_2d_rect.c......................... 206

Figure 7-5 Output of prims_2d_circle.c....................... 209

Figure 8-1 NURBS Curve Control Points and Bounding Polyhedron ... 231

Figure 8-2 NURBS Curve Knot Vector and Parameter Range......... 233

Figure 8-3 NURBS Curve Metric and Chordal Approximation Criteria . 236

Figure 8-4 Output of nurbs_bezier.c 238

Figure 8-5 Output of nurbs_circle.c 242

Figure 8-6 Untrimmed NURBS Surface 251

Figure 8-7 Trimming Curves in Surface Parameter Space 252

Figure 8-8 Trimmed NURBS Surface.............................. 253

Figure 8-9 Output of nurbs_sphere.c 260

Figure 9-1 Output of tran_2d_orig.c 278

Figure 9-2 Output of tran_2d_transl.c........................ 280

Figure 9-3 Output of tran_2d_rot.c 282

Figure 9-4 Output of tran_2d_scale.c......................... 284

Figure 9-5 Output of tran_3d.c 287

Figure 10-1 2D View Model 294

Figure 10-2 3D View Model 295

Figure 10-3 VDC Orientation 302

Figure 10-4 Output of `view_perspect.c` 306

Figure 11-1 Text Local Coordinate System 317

Figure 11-2 Annotation Text and Leader Lines 318

Figure 11-3 Character Height 321

Figure 11-4 Character Spacing 322

Figure 11-5 Character Up Vector 323

Figure 11-6 Character Slant Angle 324

Figure 11-7 Text Path 324

Figure 11-8 Internal Reference Lines of a Character 325

Figure 11-9 Examples of Text Alignment 326

Figure 11-10 Text Extent Rectangle 330

Figure 11-11 Output of `stroke_text.c` 334

Figure 11-12 Stencil Text and Block Text 339

Figure 12-1 Predefined Line Patterns 356

Figure 12-2 Line Pattern Array Formation 359

Figure 12-3 Balancing Line Patterns around a Line Segment Midpoint . . 361

Figure 13-1 XGL Predefined Markers 374

Figure 13-2 Output of `prims_2d_marker.c` 378

Figure 13-3 Output of `prims_2d_umarker.c` 380

Figure 14-1 Output of `pick_2d_prims.c` 389

Figure 15-1 Scaling Factor Applied in Scaled Depth Cueing 413

Figure 16-1 Gcache NURBS Curve and Surface Representation Modes . . 443

Figure 16-2 Output of `gcache_nsi_pgon.c` 445

Figure 16-3 Output of gcache_complex_pgon.c 447
Figure 17-1 Polygon With Texturing 481

Tables

Table 2-1 XGL Drawing Primitives 24
Table 2-2 Dimensionality of XGL Primitives 26
Table 2-3 Generic XGL Operators 28
Table 2-4 Object Relationships 30
Table 3-1 Error Categories 47
Table 3-2 Error Types 48
Table 3-3 Values of the `xgl_object_create` *type* Parameter 50
Table 4-1 Device Object Types 54
Table 4-2 Fields Returned by `xgl_inquire()` 86
Table 6-1 Objects Associated with the Context Object 161
Table 6-2 Context Object Utility Operators. 162
Table 7-1 XGL Surface Primitives. 170
Table 7-2 Geometric Primitive Operators 173
Table 7-3 General Context Rendering Attributes. 186
Table 7-4 Context Accumulation Buffer Attributes. 187
Table 7-5 Context Hidden Line and Surface Removal Attributes 188

Table 7-6	Context Line Attributes	189
Table 7-7	Context Marker Attributes	190
Table 7-8	Context NURBS Curve and Surface Attributes	191
Table 7-9	Context Edge Attributes	192
Table 7-10	Context Surface Attributes	193
Table 7-11	Context Surface Transparency Attributes	195
Table 7-12	Context Curved Surface Attributes	196
Table 7-13	Context Texture Mapping Attributes	196
Table 7-14	Context Surface Depth Cueing Attributes	197
Table 7-15	Context Stroke Text Attributes	198
Table 7-16	Context Annotation Text Attributes	199
Table 7-17	Context View and Transform Attributes	200
Table 7-18	Context Paint Type Attributes	201
Table 8-1	Providing Hints for NURBS Surface Geometry	254
Table 9-1	Transform Memberships in Matrix Groups	272
Table 10-1	Getting Context Transform Handles	296
Table 11-1	Fonts Provided by XGL	315
Table 11-2	Text Horizontal Normal Alignment Values	327
Table 11-3	Text Vertical Normal Alignment Values	327
Table 11-4	Annotation Text Attributes	329
Table 11-5	Text Encoding Schemes	331
Table 11-6	Code sets in EUC Encoding	332
Table 11-7	Possible use of Code sets in EUC Encoding	333
Table 15-1	Lighting Variables	401
Table 15-2	Lighting Parameters and Lighting Attributes	406

Table 15-3 Light Object Attributes 409
Table 15-4 Context Light Attributes 410
Table 16-1 Context Attributes for the Pcache Object 433
Table 17-1 Texture Border Conditions 456
Table 17-2 Texture Mapping in the Rendering Pipeline 462
Table 17-3 Texture Mapping Compositing Methods 462
Table 17-4 MipMap Descriptor Boundary Conditions 465
Table 17-5 Texture Sampling Methods 466
Table A-1 New Operators 493
Table A-2 Changed Attributes 494
Table A-3 Changed Data Structures 494
Table B-1 Changed Operator 495
Table B-2 Changed Operator 496
Table B-3 New Attributes 496
Table B-4 Changed Attributes 497
Table B-5 Deleted Attributes 498
Table B-6 Deleted Attributes 498
Table B-7 Deleted Attributes 499
Table B-8 New Operators 502
Table B-9 Changed Operators 503
Table B-10 Deleted Operators 505
Table B-11 New Attributes 506
Table B-12 Changed Attributes 509
Table B-13 Renamed Attributes 511
Table B-14 Deleted Attributes 513

Table B-15 New Data Structures 514
Table B-16 Changed Data Structures 517
Table B-17 Deleted Data Structures 520

Code Samples

Code Example 2-1	A First Example Program.	31
Code Example 4-1	Using XGL with Xlib.	75
Code Example 4-2	Using XGL with XView	79
Code Example 4-3	Using XGL with OLIT	83
Code Example 4-4	Inquire Example	89
Code Example 4-5	Transparent Overlay Example	100
Code Example 5-1	Simple Color Example	121
Code Example 5-2	Color Ramp Example	127
Code Example 5-3	Sharing the Window System Color Map	135
Code Example 5-4	Color Map Double Buffering Example	146
Code Example 7-1	Polygon Example	205
Code Example 7-2	Multirectangle Example	206
Code Example 7-3	Multicircle Example	210
Code Example 7-4	Example for Getting and Setting Pixels	215
Code Example 7-5	Example for Copying Pixels	217
Code Example 7-6	Accumulation Buffer Example	226

Code Example 8-1 Bezier Curve Example 238
Code Example 8-2 Circle Curve Example 242
Code Example 8-3 NURBS Surface Example 260
Code Example 9-1 Geometry Before Transformations 278
Code Example 9-2 Translation Example 280
Code Example 9-3 Rotation Example 282
Code Example 9-4 Scale Example 284
Code Example 9-5 3D Transform Example 287
Code Example 10-1 View Example 307
Code Example 11-1 Stroke Text Example 334
Code Example 11-2 Raster Text Example 342
Code Example 12-1 Line Pattern Example 364
Code Example 12-2 Edge Pattern Example 368
Code Example 13-1 Predefined Marker Example 378
Code Example 13-2 User-defined Marker Example 380
Code Example 14-1 Picking Example 390
Code Example 15-1 Facet Lighting Example 415
Code Example 15-2 Vertex Lighting Example 420
Code Example 15-3 Linear Depth Cueing Example 422
Code Example 15-4 Scaled Depth Cueing Example 425
Code Example 16-1 Gcache of Simple Polygon 445
Code Example 16-2 Gcache of Complex Polygon 447
Code Example 17-1 Texture Mapping Example 482

Preface

The *XGL Programmer's Guide* provides information on programming with the XGL™ graphics library. For additional information on writing application programs that make efficient use of XGL primitives on Sun accelerators, see the *XGL Accelerator Guide for Reference Frame Buffers*.

Who Should Use This Book

This manual is for C programmers developing XGL applications. The reader should have a basic understanding of UNIX®, the C programming language, and the OpenWindows™ windowing environment. The reader should be familiar with basic computer science concepts (such as data abstraction) and the principles of 2D and 3D computer graphics.

This manual includes discussion and example programs to assist an application programmer in writing XGL programs. It is not a comprehensive tutorial on programming with the XGL library. Examples are provided to illustrate concepts. Refer to the *XGL Reference Manual* for complete descriptions of the XGL operators, attributes, data structures, and macros. For more information on specific graphics concepts, see the references listed at the end of this preface.

How This Book Is Organized

This manual is organized into the chapters listed below. In most chapters, the example programs are located at the end of the chapter.

Chapter 1, "Introduction to XGL," provides an overview of XGL functionality and features.

Chapter 2, "Getting Started with XGL Programming," summarizes commands used to compile, link, and run XGL applications in the OpenWindows environment. The chapter also provides an overview of basic XGL concepts and includes a simple example program that introduces XGL programming.

Chapter 3, "System State Information and Generic Operators," provides XGL system state information and describes XGL generic functions.

Chapter 4, "Devices," explains XGL interaction with display devices, stream devices, and the window system.

Chapter 5, "Color," explains the XGL color model. Topics discussed are RGB and indexed color spaces, color ramps, color cubes, dithering, and color map double buffering.

Chapter 6, "Contexts," provides details about the Context object. It discusses the operators and attributes that manage graphics and environment state information.

Chapter 7, "Primitives and Graphics Context Attributes," discusses XGL drawing primitives and lists the Context attributes that define how images are rendered.

Chapter 8, "Rendering NURBS Curves and Surfaces with XGL," explains the XGL implementation of NURBS curves and surfaces.

Chapter 9, "Transforms," details the XGL implementation of matrix transformations such as scale, rotate, translate, and provides information on the transformation operators.

Chapter 10, "View Model," discusses the XGL 2D and 3D view models, and describes coordinate systems, clipping, and related transforms.

Chapter 11, "Text," discusses the XGL implementation of stroke text, details the operators and attributes for rendering stroke fonts, and discusses how to render raster text using bitmap fonts.

Chapter 12, “Line Patterns,” describes the Line Pattern object. Line Pattern objects create patterned lines and patterned surface edges.

Chapter 13, “Markers,” describes the Marker object. The operators and attributes for creating and manipulating markers are discussed.

Chapter 14, “Picking,” discusses the XGL picking implementation and describes operators and attributes for fine-tuning an application’s picking capabilities.

Chapter 15, “Lighting, Shading, and Depth Cueing,” discusses the general concepts of the 3D rendering pipeline—lighting, shading, depth cueing, and color mapping. It discusses in detail XGL-specific information such as lighting techniques, equations, operators, and attributes.

Chapter 16, “Caching Geometry,” discusses the Pcache object, which provides XGL’s display list functionality, and the Gcache object, which simplifies complex primitives to accelerate their rendering..

Chapter 17, “Texture Mapping,” presents XGL’s texture mapping functionality .

Appendix A, “Changes to the XGL 3.2 Library,” provides information on the differences between the XGL 3.2 library and the previous version.

Appendix B, “Changes to the XGL Library From XGL 3.0 through XGL 3.1” provides information on changes in the XGL product from XGL 3.0 to XGL 3.1.

Appendix C, “Software Rendering Characteristics,” provides information on rendering through XGL’s software implementation.

Appendix D, “The Utility and Main Example Programs,” lists the utility programs and main program used by the example programs in this manual.

Appendix E, “XGL Errors,” contains a list of all the XGL errors.

“A Glossary of XGL Terms” defines commonly-used XGL terms.

Related Books

- *XGL Reference Manual*
- *XGL Accelerator Guide for Reference Frame Buffers*
- Angel, E. *Computer Graphics*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.
- Bartels, R., et al. *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*, Morgan Kaufmann Publishers, Los Altos, California, 1987.
- Burger, P., and D. Gilles. *Interactive Computer Graphics*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.
- Farin, Gerald. *Curves and Surfaces for Computer Aided Geometric Design*, Second Edition, Academic Press, Inc., San Diego, CA, 1990.
- Foley, J., A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles and Practice*, Second Edition, Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.
- Gaskins, Tom. *PHIGS Programming Manual*, O'Reilly and Associates, Sebastopol, California, 1992.
- International Standard ISO/IEC 9592-4, Information Processing Systems - Computer Graphics - Programmer's Hierarchical Interactive Graphics System (PHIGS), Part 4 - Plus Lumiere Und Surfaces, February 1991.
- Newman, W., and R. Sproull. *Principles of Interactive Computer Graphics*, Second Edition, McGraw-Hill Book Company, New York, 1979.
- Watt, A. *Fundamentals of Three-Dimensional Computer Graphics*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

Ordering Sun Documents

The SunDocs℠ program provides more than 250 manuals form Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals using this program.

For a list of documents and how to order them, see the catalog section of SunExpress™ On The Internet at `http://www.sun.com/sunexpress.`

What Typographic Changes Mean

The following table describes the typographic changes used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% You have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`** `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.

New Features

This section provides general information on new features in the XGL 3.3 release.

User Control Over Edge Offset

At this release, a new attribute, `XGL_3D_CTX_EDGE_Z_OFFSET`, can be used to specify the offset added to a surface to define the DC location of the surface edge. The value specified by this attribute is a fraction of the device maximum Z coordinate. This attribute only has an effect when the Z-buffer is enabled.

Inquiring Device Transparency Support

Device support of transparency can now be inquired using the XGL operator `xgl_inquire()`. This operator will return information specifying whether transparency is accelerated in hardware or handled in software.

Introduction to XGL 1

Overview of XGL Functionality

XGL™ is a software library of two-dimensional (2D) and three-dimensional (3D) graphics primitive functions designed to support a wide variety of graphics-based applications. The XGL library provides immediate mode, nondisplay-list functionality and is suitable for supporting a number of graphics application programming interfaces (APIs), including GKS and PHIGS.

The XGL library implicitly uses hardware graphics acceleration whenever possible. Hardware graphics acceleration occurs automatically without explicitly requesting it from within an XGL program.

The XGL system requires a window system to manage drawing operations sent to a display device; it cannot render graphics on a raw display screen. An XGL application runs within a window environment managed by an X11 compatible server, such as the X11 server within Sun's OpenWindows™ environment. XGL primitives can render images onto multiple graphical display devices that coexist within the same XGL session. An example of a display device is an X11 window linked to the window system with a window handle.

The XGL system insulates the application programmer from the specifics of the window system and the underlying hardware device with XGL *objects*, which describe virtual components of the graphics rendering system. XGL objects simplify the work of the application programmer by presenting a consistent, device-independent graphics model. For example, display devices, such as X11 windows, are known to the XGL programmer as Window Raster Device

objects. Graphics state information describing how XGL graphics primitives are drawn on the device is stored in XGL 2D Context or 3D Context objects. See Figure 1-3 on page 6 for a diagram of the XGL object hierarchy.

The XGL libraries reside directly above the hardware and firmware of the display device and graphics accelerator, minimizing software overhead within the XGL graphics rendering pipeline. XGL uses Sun's Direct Graphics Access (DGA) display technology to accelerate XGL applications. DGA arbitrates access to the display screen between an XGL Window Raster and the X11 window system. DGA is available for graphics accelerators in Sun's OpenWindows environment.

The XGL library provides the applications programmer with the following graphics capabilities:

Immediate-mode rendering
Immediate-mode rendering refers to immediate rendering of graphics primitives without placing the data in a display list. XGL leaves organization of the data to the application, which permits efficient management of the graphics geometry without imposing constraints on how the geometry is stored or what specific information is kept in the display list. Immediate-mode rendering is preferable for highly interactive applications or for applications maintaining specialized data, such as those in electronic and mechanical CAD, animation, and simulation.

Loadable device pipelines
XGL 3.0 and subsequent releases support loadable device pipelines. The loadable device pipeline architecture makes it easier for hardware developers who want to port the XGL library to their graphics devices. If a hardware vendor releases a new graphics device after a release of the XGL core library and provides a library supporting the loadable device interfaces for that device, an application compiled for XGL will work without recompilation on the new device. This feature allows graphics devices to be released independently of XGL releases.

Separate 2D and 3D graphics pipelines
The XGL library supports separate, complete 2D and 3D rendering pipelines that are accessible within the same application. The 2D pipeline is optimized for 2D applications and does not carry the overhead of 3D functionality. The 3D pipeline has a rich set of capabilities for 3D applications. An application

can render 2D and 3D data to the same device. The XGL dual pipeline approach allows programmers to use either the speed of the 2D pipeline or the capabilities of the 3D pipeline.

Broad primitive and coordinate type support
The XGL library has a rich set of graphics primitives and view and modeling transforms. Standard features include 2D and 3D primitive support (such as lines and polygons); depth cueing, lighting, and shading; nonuniform B-spline curves and surfaces; and direct and indirect color model support.

NURBS surfaces
XGL supports nonuniform rational B-spline (NURBS) surfaces. This primitive provides computer-aided geometric design applications with curved surface support. The primitive follows the semantics defined by PHIGS PLUS.

Multi-primitive operators
The XGL library provides *multi-primitive* operators that permit batching of simple primitive calls into groups, based on knowledge of coordinate and attribute data. In applications such as electronic design, which repeatedly render the same type of primitive, the performance gains of batched primitive calls can be substantial.

Transparent acceleration
The XGL library provides direct mapping of graphics functionality to the underlying hardware. Where hardware acceleration does not exist, XGL provides software emulation, allowing applications to run and produce nearly identical results on all platforms and graphics devices.

Xlib and PEXlib device support for distributed networks
The XGL system supports Xlib devices (X protocol) and PEXlib devices (PEX protocol), enabling XGL programs to run across a network on remote systems. If the server supports the PEX extension and the XGL system has access to the PEXlib loadable library, XGL will emit PEX protocol that the server can interpret to generate for 2D and 3D graphics on a remote display.

Toolkit independence
XGL works with OpenWindows-supported toolkits or window managers. OpenWindows provides full X Window System capabilities and includes support for the X11 server, the X toolkits OLIT and XView™, and the Xlib library. XGL also works with most X11-compatible toolkits.

The XGL design is best suited to applications that maintain their own display list or graphics data structure. By giving applications greater control over the graphics to be rendered on the device, the XGL library provides functionality not found in PHIGS or GKS.

An application can use the XGL library as a high-level graphics library that acts as the application's interface to the hardware, or it can use the XGL library as a foundation low-level library under another high-level graphics library, such as PHIGS, as illustrated in Figure 1-1.

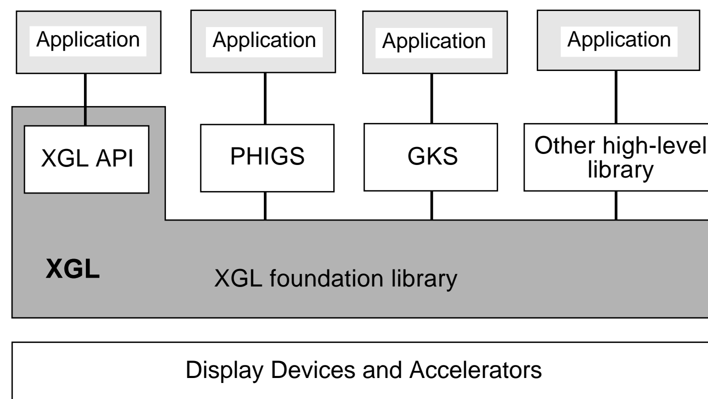


Figure 1-1 XGL API and Foundation Library

Direct Graphics Access

The XGL system uses Sun's Direct Graphics Access (DGA) technology to achieve high-performance graphics on graphics accelerators within an X11 window. As illustrated in Figure 1-2 on page 5, when a client XGL application is running locally (on the same machine as the X11 server), DGA allows the XGL library to send commands directly to the accelerator rather than sending protocol commands to the server. XGL thus avoids the overhead from building X11 protocol requests and passing them from client to server. Using a low-overhead synchronization mechanism, DGA ensures that XGL drawing operations are synchronized with the server. It also preserves the integrity of the display when windows are resized, moved, or obscured. DGA is transparent to the XGL application programmer.

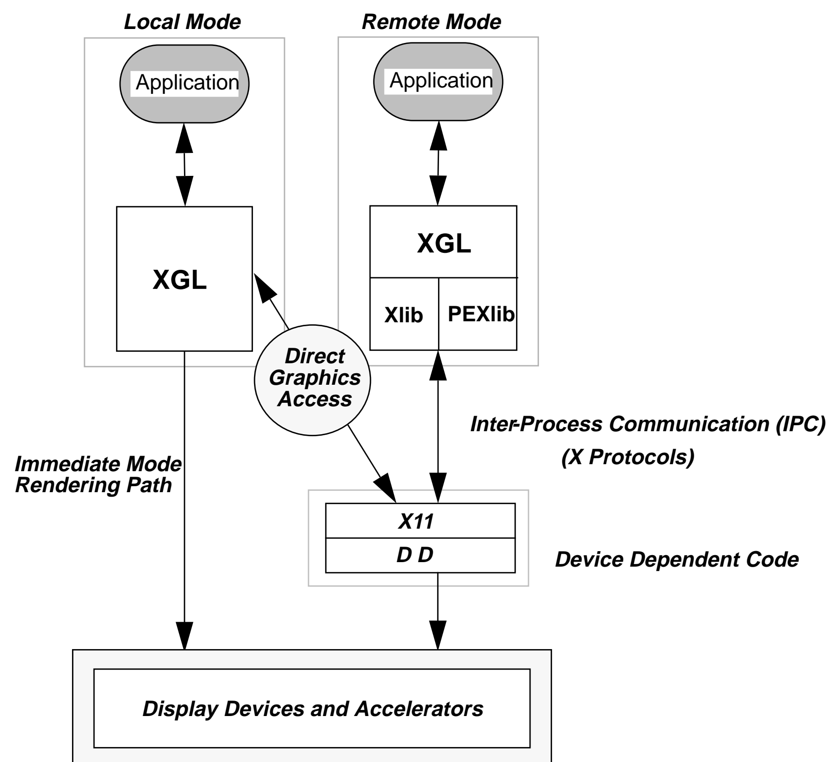


Figure 1-2 XGL in the OpenWindows Environment

Applications or new graphics interfaces layered on top of the XGL API automatically benefit from features implemented in the XGL library, such as DGA and remote rendering through X11 or PEX protocol.

Introduction to XGL Objects and Graphical Organization

The XGL API is structured hierarchically as a set of abstract data types called *classes* (see Figure 1-3). The class of an object determines the attributes it possesses and which operators can act on it. The class defined hierarchically below an existing class is a *subclass* (child) of the existing (parent) class. Applications can create *instances* of classes, where an instance of a class is an *object*. Although XGL is class-based, an application can only define instances of classes and cannot extend the hierarchy by creating new classes.

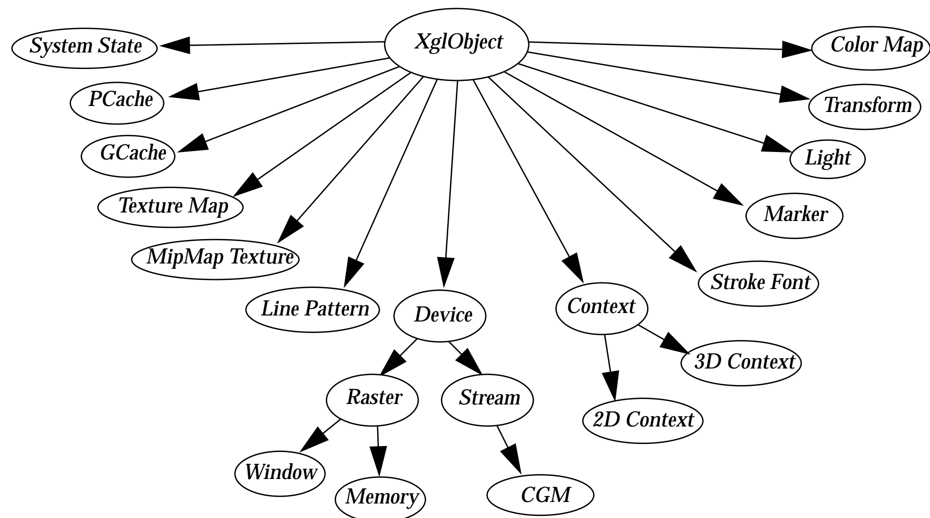


Figure 1-3 XGL Class Hierarchy

In the XGL API, an *object* is an abstraction of a graphics resource. Objects consist of *state information* (known as *attributes* in XGL) and a set of operators. Operators perform predefined actions on objects of their related class.

The XGL objects available for the manipulation of 2D and 3D data are *System State*, *Context*, *Device*, *Pcache*, *Gcache*, *Color Map*, *Transform*, *Stroke Font*, *Line Pattern*, *Light*, *Marker*, *Texture Map*, and *MipMap Texture*. Following is a brief discussion of XGL objects; more information about each object type is provided in subsequent chapters.

System State Object and Generic Operators

The System State object maintains information about all operations occurring during a single XGL session. Operators of the System State object open and close XGL and adjust the degree of error handling that XGL performs. The open and close operators, respectively, are always the first and last operators executed from within an XGL application. Only one System State object is supported for any single XGL session. See Chapter 3, “System State Information and Generic Operators” on page 43 for more information on the System State object.

The generic operators create objects, retrieve attribute information about objects, change attribute values, and destroy objects. These operators are used with all XGL objects. Figure 1-4 illustrates the generic operators and their relationship to the application and the XGL runtime system.

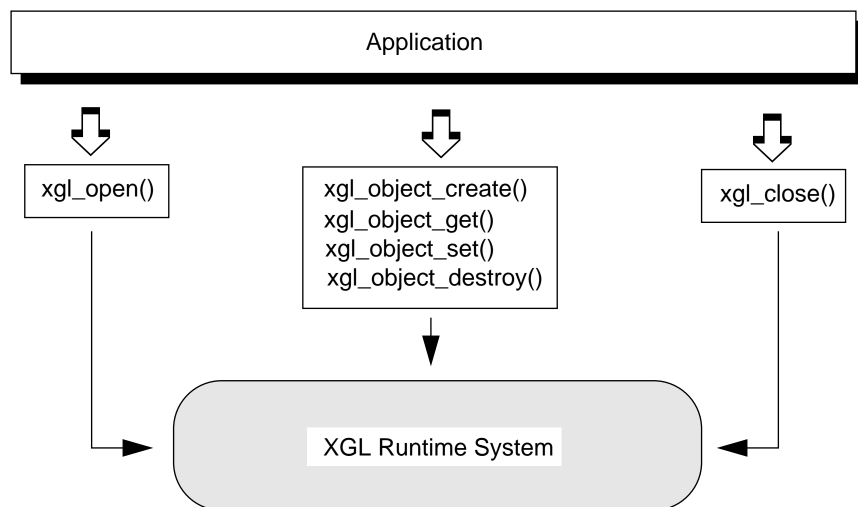


Figure 1-4 System State and Generic Operators

Device Object

The Device object maintains information directly related to the drawing surface and provides a framework for interacting with hardware graphics devices in an abstract manner. As shown in Figure 1-5, the Device class is subclassed to create a Window Raster object that renders onto a screen, a Memory Raster object consisting of an off-screen array stored in system memory, or a Stream object. A Stream device provides a protocol-independent pipeline for creating formatted output such as Computer Graphics Metafile (CGM) output. The XGL Stream device maintains no protocol itself but relies on the XGL-provided CGM device or an instantiated third-party non-raster device pipeline to implement the appropriate output protocol data format. The Device object can be attached to one or more Context objects, rendering the drawing operations executed from within the Context object on the attached device.

The Window and Memory Raster Devices are configured using the Raster Device attributes. For Window Raster Devices, the Device object maintains the connection to the windowing system. For more detailed information on XGL Device objects, see Chapter 4, “Devices” on page 53.

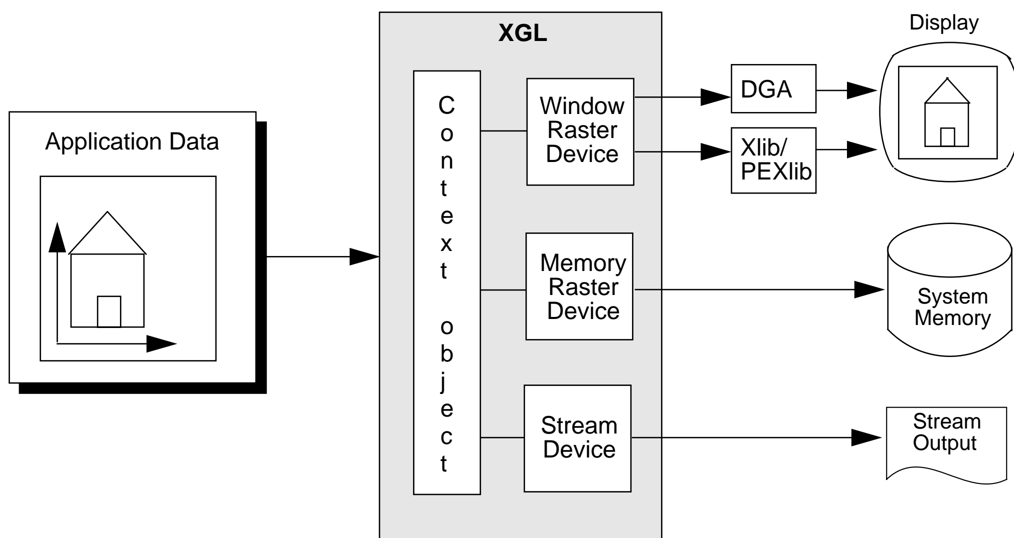


Figure 1-5 Overview of the Device Object

2D and 3D Context Objects

The 2D and 3D Context objects provide 2D and 3D drawing primitives and graphics attributes for rendering, as illustrated in Figure 1-6. The Context object also maintains graphic and environment information on coordinate systems and includes information that specifies the mapping and clipping of geometric data between coordinate spaces in the rendering pipeline. Several Context objects can exist simultaneously and can share other XGL objects as resources. 2D and 3D Context objects are distinct objects designed to meet the unique needs of 2D and 3D applications.

Primitives

XGL provides primitive operators as graphics tools for creating and manipulating geometric images. The XGL primitive operators are functions for drawing markers, polylines, curves, rectangles, arcs, circles, polygons, surfaces, and text. The characteristics of these primitives are specified in the Context object via Context attributes. For information on the Context object, see Chapter 6, "Contexts" on page 159 and Chapter 7, "Primitives and Graphics Context Attributes" on page 169.

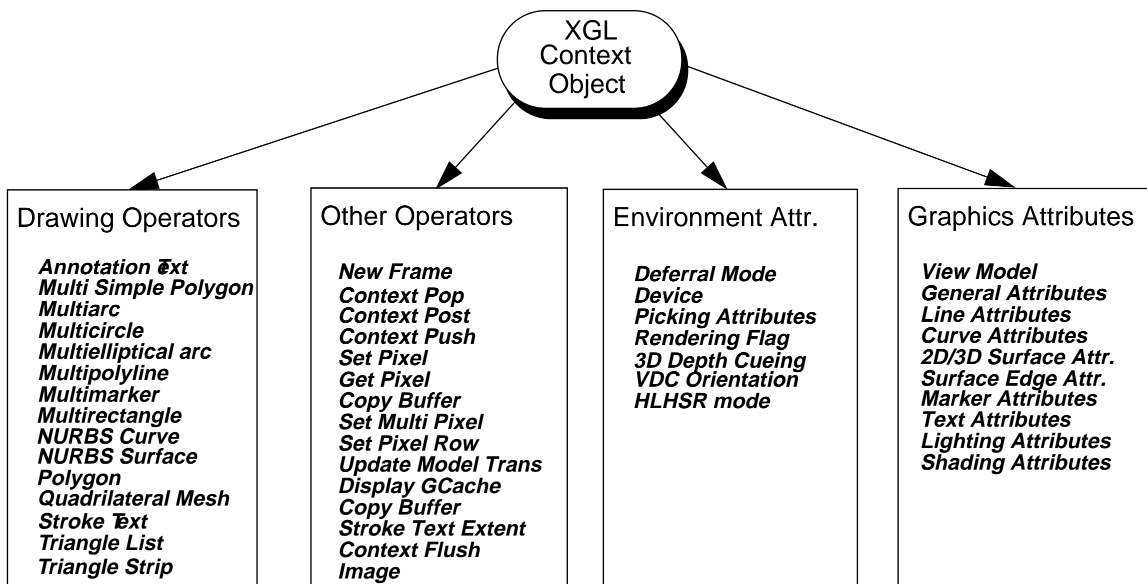
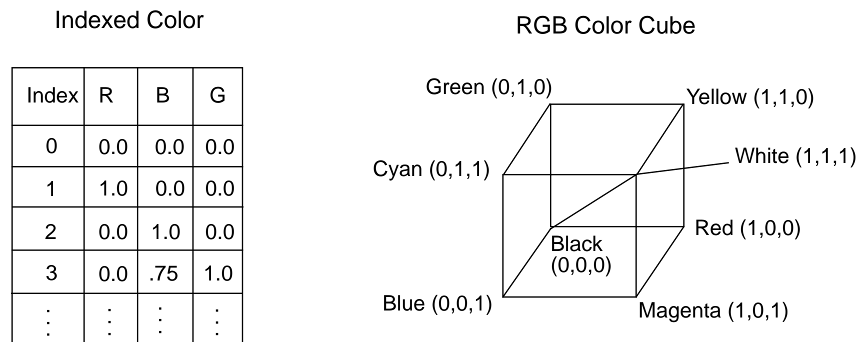


Figure 1-6 Context Object Operators and Attributes

Color Map Object

The Color Map object is an abstraction of a color table. The Color Map specifies colors in a device-independent manner and allows the XGL application to work with its preferred color model. When attached to the Device object, the Color Map object ensures optimal correspondence between the application's color model and the color model of the device.

The XGL library supports indexed and RGB color models. The Color Map object defines color value conversions from one color type to another. If the underlying hardware is indexed, the application can request RGB colors, and the Color Map object performs the RGB-to-indexed color conversion using a color cube. If the underlying hardware is RGB, the application can request indexed colors, and the Color Map object converts the color values using a color table. Figure 1-7 illustrates the two color models. Chapter 5, "Color" on page 115 provides more detailed information on the Color Map object.



Index	R	B	G
0	0.0	0.0	0.0
1	1.0	0.0	0.0
2	0.0	1.0	0.0
3	0.0	.75	1.0
⋮	⋮	⋮	⋮

Figure 1-7 Overview of the XGL Color Models

Transform Object

The Transform object provides attributes and operators for geometric transformations performed on graphics primitives. The Transform object represents a set of functions for geometric transformation operations as well as a set of transformation matrices used to map geometry from one coordinate system to another on its way through the rendering pipeline. A single 2D

Context or 3D Context object may use several Transform objects, which in turn can be shared among several Context objects. Figure 1-8 shows geometry that has been rotated and translated in 2D space using Transform object operators. Transform objects are discussed in Chapter 9, “Transforms” on page 267.

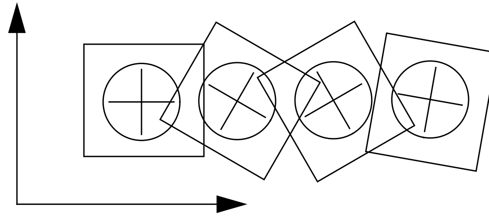


Figure 1-8 Transformed Geometry

Line Pattern Object

The Line Pattern object allows an application to specify a pattern for lines or for surface edges. The application can explicitly define the line pattern, or it can use any of the line patterns that the XGL System State object defines internally. Line Pattern objects can be shared between Context objects and may be referenced simultaneously by more than one Context object. Figure 1-9 illustrates line patterns. See Chapter 12, “Line Patterns” on page 355 for more discussion of the Line Pattern object.

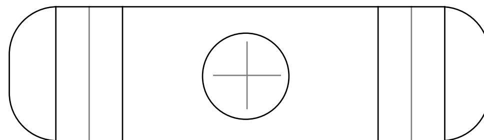


Figure 1-9 Line Pattern Example

Light Object

The Light object allows a 3D Context object to render graphics primitives with *ambient*, *positional*, *directional*, and *spot* light sources. Light objects can be shared by different Contexts while remaining independently controllable on a per-Context basis. This provides the application with flexibility, permitting individual Contexts to share some lighting attributes (such as position) but not others (such as light types enabled).

Figure 1-10 shows the four available types of light sources. Chapter 15, "Lighting, Shading, and Depth Cueing" on page 395 discusses the Light object and the XGL illumination model.

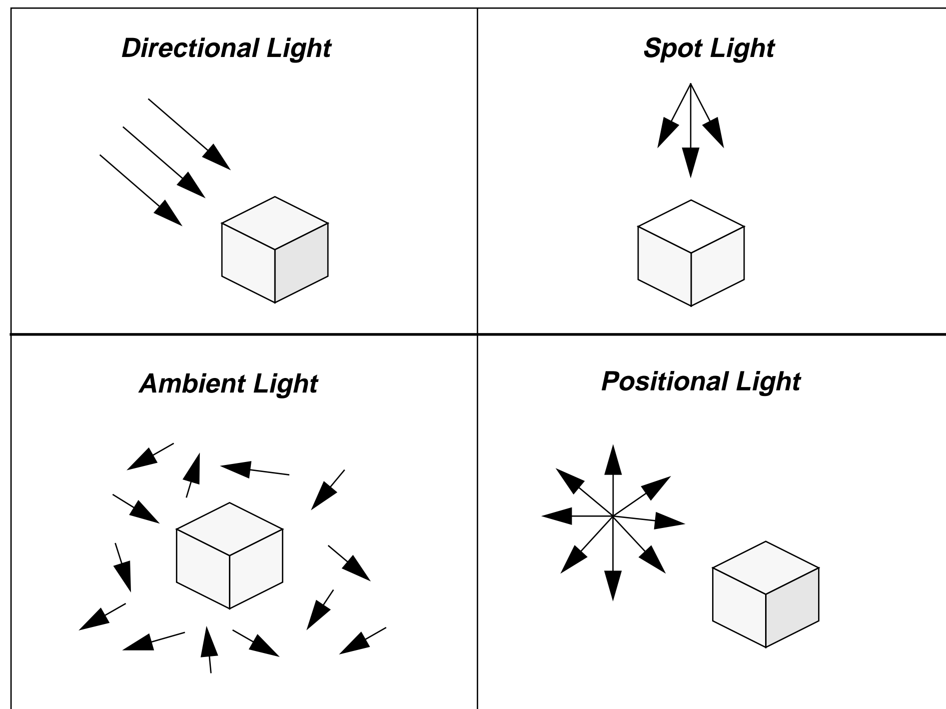


Figure 1-10 Overview of Light Sources

Stroke Font Object

The Stroke Font object allows an application to render stroke text in a variety of font types. By default, the Context object renders a monospaced Roman font. An application can use additional fonts by creating Stroke Font objects and associating them with the Context object. Stroke Font objects can be shared between Contexts and can be referenced simultaneously. Text can be rendered in any orientation in 3D space using the stroke text primitive, or it can be rendered parallel to the display surface using the annotation text primitive. Information on XGL stroke and annotation fonts is provided in Chapter 11, "Text" on page 313.

Marker Object

The Marker object allows applications to describe their own markers or symbols and share them between different Contexts. An application can define its own marker or use any of the predefined markers that the XGL library provides. A user-defined Marker can contain as many as 128 vectors. Figure 1-11 shows a spline with markers designating its control points. See Chapter 13, "Markers" on page 373 for information on the Marker object.

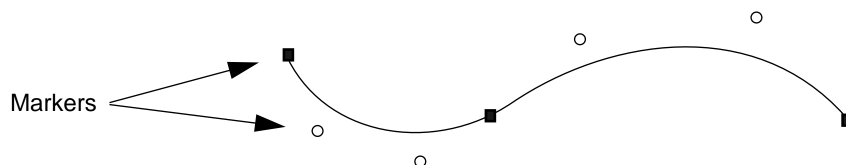


Figure 1-11 Marker Examples

Pcache Object

The Pcache object provides non-editable, non-hierarchical display list functionality in XGL. The Pcache object stores a sequence of primitives and relevant attributes for rendering at one time. Using the Pcache object, application programmers can write XGL code and tune it for the high performance that display lists can provide. For more information on Pcaches, see Chapter 16, "Caching Geometry" on page 431.

Gcache Object

Gcache objects (Geometry Cache) reduce the complexity of application geometry by processing the geometry into simpler forms. This object reduces complex primitives into many simple primitives, storing the relevant attribute values. For more information on Gcaches, see Chapter 16, “Caching Geometry” on page 431.

Texture Map Object and MipMap Texture Object

The Texture Map object and the MipMap Texture object enable applications to texture 3D surface primitives using a texturing raster image specified by the MipMap Texture object. Textures are defined in a normalized texture space (u,v), with the u and v coordinates defined in the range 0.0 to 1.0.

Mapping of the texture onto a polygon is accomplished by deriving (u,v) values from the vertex, normal, or data fields of the polygon vertices. For example, if a texture image is mapped completely onto a four-sided polygon (in a simple manner with no wrapping), the lower vertex of the polygon would be represented by the (u,v) coordinate pair (0.0,0.0), the upper left vertex by (0.0,1.0), the upper right by (1.0,1.0), and the lower right by (1.0,0.0), as illustrated in Figure 1-12.

The result of the texturing operation can be applied to different stages of the rendering pipeline. XGL supports sampling methods such as point, bilinear and trilinear to obtain texture value and supports several color composition techniques, such as blend, decal and modulate. See Chapter 17, “Texture Mapping” on page 451 for more information on texture mapping.

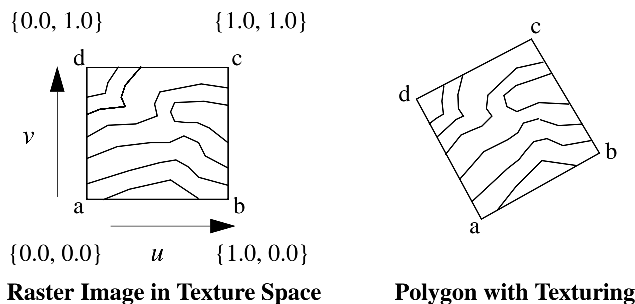


Figure 1-12 Texture Mapping of a MipMap to a Polygon

Getting Started with XGL Programming

This chapter explains how to compile an application program with XGL. It also provides information on XGL concepts and programming and illustrates these concepts with an example program.

Compiling and Running XGL Programs

To compile an XGL application program, the application program source files must reference the XGL external C header file (used to define XGL data types and structures) with the statement `#include <xgl/xgl.h>` and link with the XGL library. If the application uses a toolkit, the application must also compile and link with toolkit libraries as described in the section below. If the application will be run on Sun hardware, it must also link with the OPEN LOOK graphics libraries.

Most applications will probably be compiled using `make(1)` to save time. When creating a `Makefile` for an XGL application, note the following:

- If the XGL binaries are not in the default location of `/opt/SUNWits/Graphics-sw/xgl`, the environment variable `$XGLHOME` must be set to point to the XGL files. This environment variable can be used in an XGL `Makefile`; it specifies the location of the XGL include files (using the -I option to `cc`) and the XGL library files (using the -L option to `cc` or `ld`).
- Since the application will be running in the OpenWindows environment, the environment variable `$OPENWINHOME` should be set to point to the OpenWindows files in `/usr/openwin`.

- Applications using the XView toolkit need to link with the XView library `-lxview` and the OPEN LOOK graphics library `-lolgx`. Applications using Xlib need to link with the X library `-lX11`. Applications using the OLIT widget set need to link with the OLIT library `-lXol` and with the X Toolkit Intrinsics library `-lXt`.

Note – Sun no longer supplies compilers as part of the standard operating system distribution. To compile an XGL application, you must have an ANSI-C compatible compiler that will run on the current release of Solaris.

The application is dynamically linked with the XGL library by including `-Bdynamic` in the link line. There is no static version of the XGL library in the XGL product. The following listing shows a sample `Makefile` for an XGL application:

```
# Sample Makefile for an XGL/OpenWindows application
.KEEP_STATE:
HFILES = <application .h files>
CFILES = <application .c files >

XGLHOMELIB:sh=echo ${XGLHOME-/opt/SUNWits/Graphics-sw/xgl}
OPENWINHOMELIB:sh=echo ${OPENWINHOMELIB-/usr/openwin}
OFILES = $(CFILES:%.c=%.o)
XGL_LIBS = -lxgl
SYS_LIBS = -lXol -lXt -lxview -lolgx -lX11 -ldga -ldl
OS_LIBS_5 = -lintl -lnsl -lsocket
OS_LIBS = $(OS_LIBS_$(OSVER))

LIBS = -L$(OPENWINHOMELIB)/lib -L$(XGLHOMELIB)/lib $(XGL_LIB) \
        $(SYS_LIBS) $(OS_LIBS) -lm
CFLAGS = -O
CPPFLAGS = -I$(XGLHOMELIB)/include -I$(OPENWINHOMELIB)/include

application: $(OFILES)
    $(LINK.c) $(OFILES) -R $(XGLHOMELIB)/lib:$(OPENWINHOMELIB)/lib \
    -Bdynamic $(LIBS) -lm -o application
```

Note that if the `$XGLHOME` variable is set, the `Makefile` will use that value. If the variable is not set (in other words, the XGL library is in the default location), the `Makefile` will use the default value.

Note – The XGL library runs within the current version of the Solaris product. This means that XGL is compatible with the component parts of the Solaris product. XGL is not supported with earlier versions of Solaris or its component parts.

Run-time Considerations

At runtime, the runtime loader needs to know the location of the XGL dynamic library `libxgl.so.3`. The application can pass this information to the loader either by linking the application at link time using the `-R` option or by setting the `LD_LIBRARY_PATH` environment variable to include the path to the XGL library. The link line of the sample `Makefile` above shows the use of the `-R` option. If a `Makefile` like the sample `Makefile` is used to compile an application, the `LD_LIBRARY_PATH` environment variable is not needed at runtime because the `Makefile` takes into account the values of `$XGLHOME` and `$OPENWINHOME`.

If the application was not compiled with the `-R` option, the user will need to set the `LD_LIBRARY_PATH` variable. The following example line from a user's `.cshrc` file or `.login` file shows the `LD_LIBRARY_PATH` variable being set to `$XGLHOME/lib` for the XGL library:

```
setenv LD_LIBRARY_PATH $OPENWINHOME/lib:$XGLHOME/lib
```

For more information on the `LD_LIBRARY_PATH` environment variable, see the `ld` (3) manual page.

Error Message Files and Font Files

The XGL runtime system looks in `$XGLHOME/lib/locale/en_US/LC_MESSAGES` for translated error message files and device error message files. It looks in `$XGLHOME/lib/xglfonts/stroke` for its stroke font files. If `$XGLHOME` is not set, the runtime system looks in `/opt/SUNWits/Graphics-sw/xgl/lib/locale/en_US/LC_MESSAGES` for the error files and in `/opt/SUNWits/Graphics-sw/xgl/lib/xglfonts/stroke` for the stroke font files. The search path for the font files can be changed by the application. See page 44 for information.

XGL Loadable Pipelines

XGL renders images via loadable pipelines. The loadable pipelines are shared object library modules containing device-specific functions. Although `libxgl` is loaded by the runtime loader, the loadable pipeline shared object files are loaded explicitly by a call in the XGL code. The loadable pipelines are located in `$XGLHOME/lib/pipelines` if they have been installed. If `$XGLHOME` is not set, then installed pipelines are assumed to be in the default area of `/opt/SUNWits/Graphics-sw/xgl/lib/pipelines`.

Note – If the application is running remotely and the server has loaded the PEX extension, `XGLHOME` is not used to load the device pipelines, but it is used to load the font files and error message files.

XGL Example Programs

The XGL product includes example programs that use the XGL library to illustrate basic XGL concepts. These programs are described throughout this manual. Note that since XGL has been designed to run in the OpenWindows windowing environment, all the example programs in this manual run within this environment.

If You Experience Poor Performance for Local Rendering

If you notice poor performance when rendering locally on an accelerated graphics device, the cause may be a security feature in Solaris that governs frame buffer access by processes that were started by users other than the first user of the window system. As a result of this security feature, non-owners of the window system do not have access to DGA. For example, if a user starts the window system, and then another user sits at the workstation, changes to his own environment using `su`, and starts an XGL application, the application will not run via DGA even though the second user is running the application locally. In this case, XGL uses Xlib to render even when XGL is running locally.

To give all local users access to DGA, follow these steps:

1. **Become superuser.**
2. **Change directory to the file `/etc/logindevperm`.**

3. Edit the permissions in the following lines to allow world read/write access.

```
/dev/console   0666    /dev/mouse;/dev/kbd
/dev/console   0666    /dev/sound/*        # audio devices
/dev/console   0666    /dev/fbs/*          # frame buffers
```

Use this command:

```
host% chmod 666 /dev/mouse /dev/kbd /dev/sound/* /dev/fbs/*
```

4. Reboot.

Note that the system is no longer secure.

Rendering Remotely Through PEX

If a window raster is created on a remote server that supports PEX, but the PEXlib library, `libPEX5.so`, cannot be found on the local machine, XGL issues an error message and uses Xlib to communicate with the remote window raster. To eliminate the error, either install the PEXlib library or disable PEX on the remote server.

Basic XGL Concepts

The XGL programming model and the relationship between XGL and the X window system were briefly introduced in Chapter 1. The sections that follow provide additional information on XGL and the X window system, and describe more fully the XGL object-based programming model. For an example of an XGL program, turn to "Example Program" on page 30.

XGL and the X Window System Environment

An XGL application is an X client program running in an X window system environment, such as Sun's OpenWindows environment. The X window system is composed of several components. The central component is the display server, which is responsible for managing output requests from client applications to draw onto the display and for distributing input events to the appropriate client applications. The server and the client applications communicate via the X and/or PEX protocol(s). During a work session, client applications generate protocol queries and requests using the Xlib or PEXlib programming interface or an X toolkit, and the server processes these requests.

An XGL application program uses Xlib or X toolkit functions to connect to the server, create a window on the display, and handle event interpreting. The server manages the relationship among the various windows on the screen and provides input and events to the application, and the toolkit provides the application's user interface. The application renders its geometric data to the window using XGL primitives. The Xlib or X toolkit calls coexist with the XGL calls, sharing the same process space and drawable area on the screen. Figure 2-1 illustrates a high-level view of this relationship.

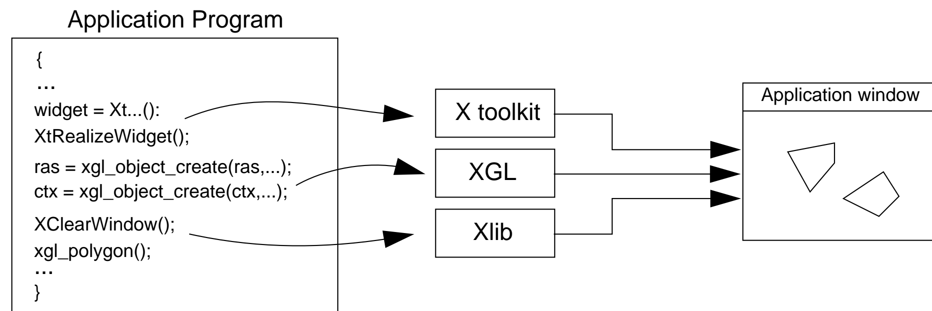


Figure 2-1 High-level View of an XGL Application Program

Rendering Graphics Locally and Remotely

When a client XGL application is running locally under Sun's OpenWindows server and the hardware supports Direct Graphics Access (DGA) protocol, XGL renders using DGA technology rather than rendering through the server via the X protocol. DGA is a set of mechanisms that enables OpenWindows client processes to directly drive a graphics device that is under the control of the OpenWindows server. Although the server manages the resources of the device, the XGL client process coordinates with the server to send rendering commands directly to the device rather than sending X11 protocol messages to the server. This allows XGL to eliminate the communication overhead of the server-based window system, thereby improving performance.

Figure 2-2 on page 22 illustrates the XGL system's relationship to DGA and the server. When an application program creates a Raster Device object, the XGL library automatically determines whether DGA is available and whether XGL supports the frame buffer. If DGA is available and the frame buffer includes DGA support, XGL synchronizes with the server to share direct access to the frame buffer. In this case, XGL renders graphics geometry to the frame buffer, and the server performs other window operations. Note that XGL will render through Xlib or PEXlib when used with a non-OpenWindows server. This allows XGL to render to any X11 server; however, rendering will be accelerated on a machine running the OpenWindows server.

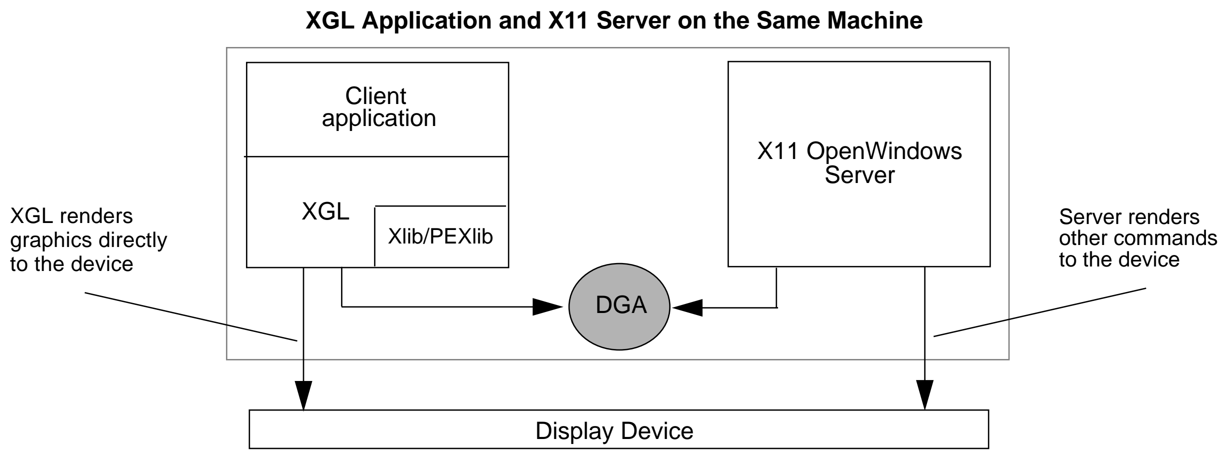


Figure 2-2 Using DGA to Render Locally

Like all X applications, an XGL application can be run remotely and displayed on a user's local workstation if the two workstations are part of a network. Remote rendering through the X11 protocol or the PEX protocol is handled automatically by XGL. When the XGL client program is running remotely, XGL uses Xlib or PEXlib to do all rendering. If the server includes the PEX extension and XGL has access to its PEX loadable library, XGL uses PEXlib to render. If PEX is not available, XGL uses Xlib for 2D rendering. For primitives and rendering options that are not supported by Xlib, XGL does scan conversion to send the pixels through Xlib to the device. Figure 2-3 on page 23 illustrates remote rendering.

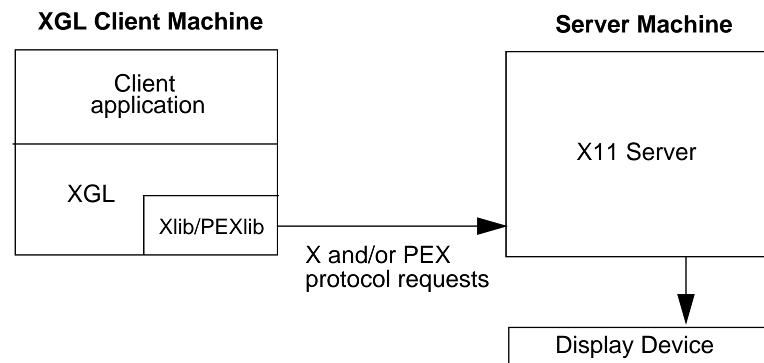


Figure 2-3 XGL and Remote Rendering

How an XGL Application Works

As mentioned above, an XGL application follows the general format of Xlib, PEXlib, or X toolkit programming, using the event-driven model of X applications for interaction handling. With Xlib calls, PEXlib calls, or X toolkit calls, the application first creates the window system objects that it needs, such as a window for graphics display, and then opens the XGL system. When XGL is opened, it automatically creates the System State object as well as internal objects that handle interactions between the device-dependent and the device-independent parts of the XGL system. To set up a framework for rendering, the application must create a Device object, which is an abstraction representing the display device, and a Context object, which controls all rendering actions on a device. These objects are created using an XGL object creation operator that takes attributes describing the object as input arguments, creates an instance of the object, and returns a handle to the object. The application program must associate the Device object with the Context object before geometry can be rendered. When the Context object and Device object are associated, the application can use XGL primitives to render geometry and can pass control of the program to Xlib or an X toolkit to process events.

During the work session, the application can change Context attributes to change the display characteristics of geometry. It can also render geometric data using XGL drawing primitives and create other objects as needed. For example, the application might want to create several Stroke Font objects to enable the use of different character sets, or create Line Pattern objects to provide the user with application-specific line patterns. Multiple Device

objects, such as Window Raster and Memory Raster devices, can be associated with and disassociated from the Context object as needed for rendering. When the application exits XGL, the System State object destroys existing objects, frees resources, and then destroys itself, closing XGL.

XGL Drawing Primitives

The XGL library provides a set of drawing primitives that the application can use to render geometric data. The available primitives include basic line, polygon, and text primitives as well as more complex primitives, such as NURBS curve, curved surface, and quadrilateral mesh primitives. In the XGL system, geometry is rendered via the Context object, which also maintains graphic and environment state information that is used in rendering. Because XGL is an immediate mode system, it needs access to current state information at the time of rendering. Since the application can create more than one Context object and can render the same geometric data using different Context objects, the application must define which Context object is to be used to render. Thus, the drawing primitives require the Context object as an input parameter and are considered to belong to the Context object.

The drawing primitives are listed in Table 2-1 and are described in more detail in Chapter 7, “Primitives and Graphics Context Attributes”.

Table 2-1 XGL Drawing Primitives

Primitive Type	Description	XGL Primitive
Lines	A set of unconnected lines	`xgl_multipolyline()`
B-spline curves	A spline (NURBS) curve	`xgl_nurbs_curve()`
Markers	A set of markers	`xgl_multimarker()`
Circles	A set of circles	`xgl_multicircle()`
Circular arcs	A set of arcs	`xgl_multiarc()`
Elliptical arcs	A set of 3D elliptical arcs	`xgl_multi_elliptical_arc()`
Rectangles	A set of rectangles	`xgl_multirectangle()`
Polygon	A single planar polygon	`xgl_polygon()`
Multiple polygons	A set of 3- or 4-sided polygons	`xgl_multi_simple_polygon()`
Stroke text	A text string	`xgl_stroke_text()`

Table 2-1 XGL Drawing Primitives

Primitive Type	Description	XGL Primitive
Annotation text	A text string displayed in a plane parallel to the display surface	`xgl_annotation_text()`
Quadrilateral mesh	A set of connected quadrilateral polygons	`xgl_quadrilateral_mesh()`
Triangle strip	A set of connected triangular polygons	`xgl_triangle_strip()`
Triangle list	A set of connected triangles arranged as a triangle strip or as a triangle star, or a set of unconnected triangles	`xgl_triangle_list()`
Curved surface	B-spline surface	`xgl_nurbs_surface()`

Handling 2D and 3D Data

XGL handles the dimensionality of application data via the dimension of the Context object, the type of point data input to the drawing primitives, and the set of attributes available to a Context. Context objects can be either 2D or 3D. An application that needs to render both 2D and 3D model data will typically create two Context objects, one for 2D rendering and one for 3D rendering. When the Context object is created, a default rendering pipeline, which includes transformations and clipping,, is also created. Creation of a 2D Context object results in the creation of a pipeline tailored to the processing of 2D geometry; creation of a 3D Context results in the creation of a more complex 3D pipeline, which handles lighting, shading, and texturing.

Application model data must be input to the drawing primitives using a point type appropriate to the Context object. Thus, for example, a 2D application might define a 2D Context object and set up the application data in XGL 2D point structures, choosing from integer or floating point structures. The XGL library provides a wide variety of point data types, including point types with and without color and/or normal data.

The dimension of the Context object also controls the set of attributes available to the application for rendering. For example, a 2D Context object includes a set of attributes that controls how the front of a surface is displayed; a 3D Context object has access to the front surface attributes but has an additional set of attributes for back surface rendering as well.

Once the appropriate Context object has been created, the application model data has been set up in XGL point data structures, and the applicable attributes have been set, the application can render using the drawing primitives. Many of the XGL drawing primitives can be used for both 2D and 3D rendering, but some are specific to 3D geometry, as shown in Table 2-2.

Table 2-2 Dimensionality of XGL Primitives

Primitive	**2D Rendering**	**3D Rendering**
`xgl_multipolyline()`	✓	✓
`xgl_multimarker()`	✓	✓
`xgl_multicircle()`	✓	✓
`xgl_multiarc()`	✓	✓
`xgl_multi_elliptical_arc()`		✓
`xgl_multirectangle()`	✓	✓
`xgl_polygon()`	✓	✓
`xgl_multi_simple_polygon()`	✓	✓
`xgl_stroke_text()`	✓	✓
`xgl_annotation_text()`	✓	✓
`xgl_quadrilateral_mesh()`		✓
`xgl_triangle_strip()`		✓
`xgl_triangle_list()`		✓
`xgl_nurbs_curve()`	✓	✓
`xgl_nurbs_surface()`		✓

More on XGL Object-Based Programming

XGL represents graphics information with abstract data structures called objects. An XGL object represents a graphics resource, such as a line pattern or a font. Each object describes a virtual component of the graphics rendering system and contains information necessary to perform graphics operations. A display device, for example, is known to the XGL programmer as a Window Raster Device object. This abstraction of the display device hides the specifics of the device-dependent code for the graphics hardware device and lets the XGL programmer interact with the device in a device-independent manner.

Internally, XGL objects are instances of classes. A class is an abstract data type that combines state information with functions that perform actions on the object. The class's attributes and functions define the characteristics of the object. A class is only a data type definition; it does not exist in memory. A class must be instantiated to be used, and it can be instantiated many times to create distinct objects that all belong to the same class. Objects exist in system memory and provide the functionality that the application program needs to render geometry.

Figure 2-4 on page 28 shows a diagram of the XGL class hierarchy with objects that have been instantiated by calls to `xgl_object_create()`. The illustration shows a possible XGL runtime system consisting of two Context objects, a Window Raster object, a Memory Raster object, two Stroke Font objects, and a Transform object.

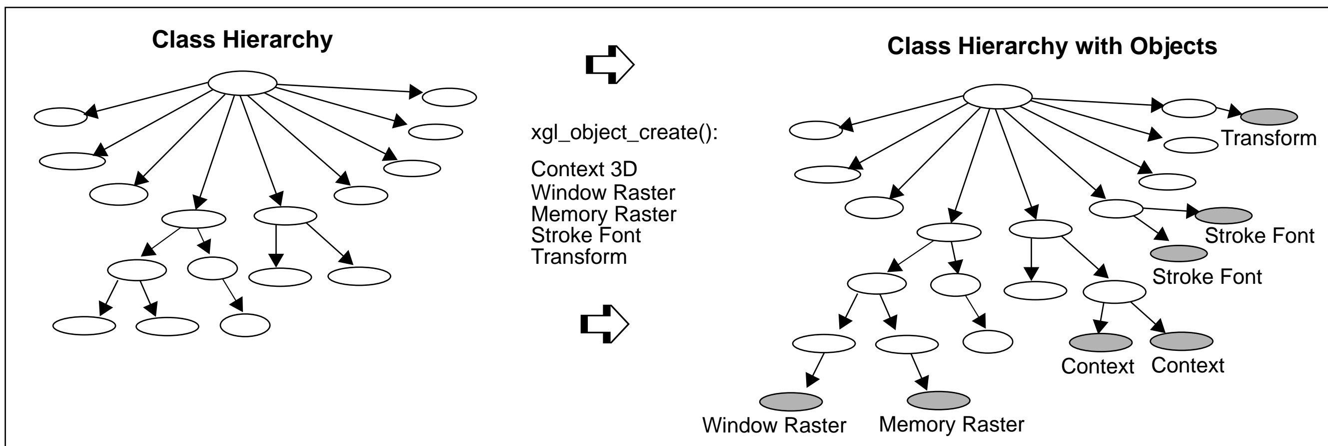


Figure 2-4 Instantiated Objects

Operators

Operators are functions that control the behavior of an object. Most XGL objects have a set of operators that handle operations required by the object. For example, the Transform object includes operators that perform transformation operations, such as matrix concatenation. The XGL system also provides operators that perform functions common to objects, such as object creation or destruction. These operators, listed in Table 2-3, provide a consistent method of working with all XGL objects.

Table 2-3 Generic XGL Operators

Operator	Description
`xgl_object_create()`	Creates an XGL object.
`xgl_object_set()`	Sets the value of an attribute.
`xgl_object_get()`	Gets the value of an attribute.
`xgl_object_destroy()`	Destroys an object.

Operators provide a way for the application to access an object and manipulate the object's data structures, since direct access to the object data structures is not provided. If, for example, the application program needs to change the

color of a line, it must use the `xgl_object_set()` operator to change the value of the line color attribute, since it cannot directly access the line color attribute field in the object data structure.

Attributes

Attributes control much of the functionality of XGL, such as the appearance of rendered geometry, the way picking is handled, and the orientation of virtual device coordinate space. Although some attributes are read-only, an application can change the value of most attributes at any time. Attributes are input as arguments to XGL operators, some of which take an attribute-value list as one of the input parameters. Each attribute-value pair in the list consists of an attribute name followed by its value. The attribute-value list may have as few as zero attribute-value pairs or as many as the total number of the object's attributes, but in all cases it is terminated with the `NULL` character, where `NULL` is defined as in `stdio.h`.

Object Relationships

Some objects use other objects as resources. When an association has been established between two objects, the objects communicate via messages that inform the using object of changes that have occurred in the data structures of the used object. For example, if an application has created a new Line Pattern object and has associated the Line Pattern object with the Context object, the Context object will automatically be kept up to date on the characteristics of the Line Pattern object so that the line pattern will be rendered correctly.

Object relationships are set up by the application and are managed internally by XGL object management functions. When an application has created a new object, it establishes the association between the new object and an existing

object using the xgl_object_set() operator, a connecting attribute, and the handles of the two objects. Table 2-4 lists objects that are associated as graphics resources with the Context and Device objects.

Table 2-4 Object Relationships

Using object	Object being used
Raster Device	System State Color Map
Context	System State Device Data Map Texture Texture Map Transform Line Pattern Marker MipMap Texture Pcache Stroke Font Light (3D Context only)

Information on the specific attributes that associate resource objects with the Context or Device objects is available in the chapters that follow.

Example Program

The hello_world.c program is a simple XGL program using the OLIT toolkit. The program opens XGL, creates a Window Raster Device object, renders simple text on the Raster in the default font, and closes XGL. The program's main() function follows these general steps:

- Initialize the OLIT toolkit and the X Toolkit internals, opening the connection to the server.
- Create the toplevel shell and the pane widget that XGL will render into, registering a callback function for expose events on the pane widget.
- Realize the widgets.
- Initialize XGL, creating the System State object.
- Create a Window Raster Device object associated with the pane.

- Create a 2D Context object and associate the Raster object with the Context object.
- Start the Xt main event loop. Once the main event loop has started, Xt handles program execution.

The output of `hello_world.c` is shown in Figure 2-5, and the complete program listing follows. Turn to page 34 for a more detailed discussion of the program. The code for this program is located in `/opt/SUNWsdk/sdk_2.6/xgl/demo/examples`, with the other XGL sample programs. To compile the program, type `make hello_world` in the example programs directory.

Figure 2-5 Output of `hello_world.c`

Code Example 2-1 A First Example Program

```
/*
 * hello_world.c
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/Shell.h>
#include <Xol/OpenLook.h>

#include <Xol/DrawArea.h>

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

```
/* Global variables */
static Xgl_object       ctx = NULL;     /* XGL context object */
static Xgl_object       sys_state;
static Xgl_object       win_ras = NULL; /* XGL window raster */

main (
    int                 argc,
    char                *argv[])
{
    static void         redraw();

    XtAppContext        app;

    Widget              toplevel, pane;

    Display             *display;     /* pointer to X display */
    Window              xwindow;      /* XID of window */
    Xgl_X_window        xgl_x_win;    /* XGL-X data structure */
   Xgl_obj_desc       win_desc;     /* XGL window raster structure */

    OlToolkitInitialize((XtPointer)NULL);
    toplevel = XtVaAppInitialize(&app, "XGL Hello",
        (XrmOptionDescList)NULL, 0,
        &argc, argv, (String *)NULL,
        XtNtitle,       "XGL Hello World Program",
        XtVaTypedArg, XtNvisual, XtRString,
        "PseudoColor", sizeof("PseudoColor"),
        NULL);

    pane = XtVaCreateManagedWidget( "pane",
        drawAreaWidgetClass,    toplevel,
        XtNborderWidth,         1,
        XtNwidth,               500,
        XtNheight,              200,
        NULL);
    XtAddCallback(pane, XtNexposeCallback, redraw, NULL);

    XtRealizeWidget(toplevel);

 /* get X stuff */
    display = (Display *) XtDisplay (pane);
    xwindow = XtWindow (pane);

    /* put X stuff into XGL data structure */
    xgl_x_win.X_display = (void *) display;
    xgl_x_win.X_window  = (Xgl_usgn32) xwindow;
```

```
    xgl_x_win.X_screen  = (int) DefaultScreen (display);

/*
 *   Open XGL, create a Raster and a Context.
 *   Attach the raster as the Context's Device.
 *   Initialize the context text attributes to render the text.
 */
    sys_state = xgl_open (NULL);

    win_desc.win_ras.type = XGL_WIN_X;
    win_desc.win_ras.desc = &xgl_x_win;
    win_ras = xgl_object_create (sys_state, XGL_WIN_RAS,
                                 &win_desc, NULL);

    /* create XGL graphics Context object using the
     * Window Raster object */
    ctx = xgl_object_create (sys_state, XGL_2D_CTX, NULL,
                       XGL_CTX_DEVICE, win_ras,
                       XGL_CTX_DEFERRAL_MODE,XGL_DEFER_ASAP,
                       XGL_CTX_STEXT_CHAR_HEIGHT, 20.0,
                       NULL);

     XtAppMainLoop(app);
}

static void
redraw()
{
    static Xgl_pt_f2d   text_pos; /* start pos of text in window */

    text_pos.x = 90.0;
    text_pos.y = 100.0;

    if (ctx) {
        xgl_window_raster_resize(win_ras);
        /* clear the display */
        xgl_context_new_frame (ctx);

        /* draw the stroke text */
        xgl_stroke_text (ctx, "Hello XGL World", &text_pos, NULL);
    }
}
```

Creating the Window and Registering Callback Procedures

The `main()` routine of `hello_world.c` initializes the OLIT toolkit, the Xt toolkit, and XGL, and opens the connection to the server. It creates the OLIT widgets that the application needs, in this case, a DrawArea widget for the pane, registers a callback procedure to handle expose events on the pane, and creates a window for the widget. It also creates the XGL objects that the application needs. Finally, it starts the main event loop to wait for X events.

The program creates a drawing surface 500 units wide and 200 units high. The window frame is labeled "XGL Hello World Program". The visual type of the window is PseudoColor.

Opening XGL

The `xgl_open()` call initializes the XGL environment.

```
sys_state = xgl_open(NULL);
```

The `xgl_open()` operator creates and returns the System State object. The System State object handles the creation of other XGL objects in subsequent `xgl_object_create()` calls.

Creating a Window Raster Device Object

To create a Window Raster Device object, the application must provide XGL with information about the X11 window with which the Raster object is associated. The application must get a pointer to the display, the X handle to the window, and an integer value identifying the screen that XGL will render to. This information is stored in an XGL window descriptor data structure that is passed to `xgl_object_create()` when the Window Raster object is created. The following code fragments show the Xt calls that retrieve window information, the XGL window data structure, and the `xgl_object_create()` call.

```
/* get X stuff */
   display = (Display *) XtDisplay(pane);
   xwindow = XtWindow (pane);
          .
          .
          .
   /* put X stuff into XGL data structure */
```

```
xgl_x_win.X_display = (void *) display;
xgl_x_win.X_window = (Xgl_usgn32) xwindow;
xgl_x_win.X_screen = (int) DefaultScreen(display);
       .
       .
       .
win_desc.win_ras.type = XGL_WIN_X;
win_desc.win_ras.desc = &xgl_x_win;
win_ras = xgl_object_create(sys_state, XGL_WIN_RAS, &win_desc,
                          NULL);
```

The `xgl_object_create()` operator returns a handle to the Window Raster Device object. The System State object stores a pointer to the Device object in its list of API objects.

Creating a Context Object

The Context object keeps track of attribute information for the XGL primitives. In this example, a 2D Context object is created by a call to `xgl_object_create()` with a *type* value of `XGL_2D_CTX`.

```
ctx = xgl_object_create(sys_state, XGL_2D_CTX, NULL,
                       XGL_CTX_DEVICE, win_ras,
                       XGL_CTX_DEFERRAL_MODE, XGL_DEFER_ASAP,
                       XGL_CTX_STEXT_CHAR_HEIGHT, 20.0,
                       NULL);
```

The Context's deferral mode is set so that XGL will render a primitive as soon as it is received. The character height for the stroke font is set to 20.0 units. The previously created Device object is associated with the Context object with the `XGL_CTX_DEVICE` attribute and the Device object's handle. The newly created Context object is registered in the System State object passed to the `xgl_object_create()` operator.

Rendering Geometry

The `redraw`() procedure renders the geometry data that the application supplies. In this program, the `redraw`() procedure clears the display using the XGL operator `xgl_context_new_frame()` and then draws text into the window with the primitive `xgl_stroke_text()`. The `xgl_stroke_text()` primitive renders the string "Hello XGL World" to the text position, using the text attributes in the Context object.

```
static void
redraw()
{
    static Xgl_pt_f2dtext_pos;/* start pos of text in window */

    text_pos.x = 90.0;
    text_pos.y = 100.0;

    if (ctx) {
        xgl_window_raster_resize(win_ras);
        /* clear the display */
        xgl_context_new_frame (ctx);

       /* draw the stroke text */
        xgl_stroke_text (ctx, "Hello XGL World", &text_pos, NULL);
    }
}
```

Objects Provided at XGL Initialization

A set of XGL objects is created during the initialization of the XGL library. Although the application must create the Context object and the Device object, other objects are automatically created, as follows:

- The System State object is automatically created during the `xgl_open()` call. The System State object in turn creates the following objects:
  - Line Pattern objects that supply the application with a set of predefined line patterns. These predefined Line Pattern objects cannot be modified by the application. To create additional Line Patterns, the application must create additional Line Pattern objects.
  - Marker objects that provide predefined symbols. Like the predefined Line Pattern objects, these Marker objects cannot be modified by the application. To create additional markers, the application must create additional Marker objects.
- When the application creates the Device object, the XGL system creates a default two-color (black and white) Color Map object. This Color Map cannot be modified. To render in color, the application must create its own Color Map object and associate it with the Device object.

- When the application creates a Context object, the following objects are also created:
  - Transform objects for transformation matrices used by the viewing pipeline. These Transform objects are the Local Model Transform, the Global Model Transform, the Model Transform, the View Transform, the VDC Transform, and for 3D Contexts, the Normal Transform. These Transforms are set to identity at initialization, but the application can retrieve several of the matrices and manipulate the values using the Transform operators. Applications can also create new Transform objects as necessary.
  - A Stroke Font object that defines the Roman_M font in the XGL font directory as the default font. To use different font, the application must create a new Stroke Font object, set it to the appropriate font name, and associate it with the Context object.

Note that no Light objects are created at initialization, but Context environment attributes can create an array of lights whose values can be retrieved and set.

Programming Tips

This section provides tips on writing XGL programs. The first group of tips contains general information and considerations about writing XGL programs. The second group of tips contains information about using XView/OLIT and XGL. Finally, XGL limitations are briefly listed.

General Tips

- The default value of the Context attribute `XGL_CTX_DEFERRAL_MODE` is `XGL_DEFER_ASTI` ("At Some TIme"). In this mode, the application must call `xgl_context_post()` to ensure that all the data is flushed from the XGL internal buffer and displayed.
- Assuming that an application has obtained X information on the X11 display, the frame window containing the drawing window, and the canvas window that XGL will draw to, the application can use the following lines of code to enable the window manager to install an application color map for the canvas window.

```
Atom       catom;
catom = XInternAtom(display, "WM_COLORMAP_WINDOWS", False);
XChangeProperty(display, frame_window, catom, XA_WINDOW, 32,
                PropModeAppend, &canvas_window, 1);
```

- `xgl_close()` should be called to ensure that all resources are released.

XView Tips

- The XView attribute `CANVAS_AUTO_CLEAR` should be set to `FALSE`, since the XGL `xgl_context_new_frame()` operator can clear the canvas (using DGA) faster than Xlib can. `xgl_context_new_frame()` also performs other actions, such as clearing the Z-buffer.
- The application should supply an event procedure to handle resize and repaint events for the canvas paint window of the XView canvas. When a `WIN_RESIZE` event occurs, the XGL `xgl_window_raster_resize()` operator should be called. If there is more than one XGL window raster in an application, the application must determine which *Xgl_win_ras* object corresponds to the *Xv_window* object passed to the event procedure.

  In addition, before calling `xgl_window_raster_resize()`, the application should call `XSync()` if it will be running as a DGA application. The XGL call `xgl_window_raster_resize()` uses DGA to grab window size information. Because of the way XView handles geometry management, the window size returned is the previous window size. Calling `XSync()` before `xgl_window_raster_resize()` will synchronize the server and XGL.
- The XView attribute `CANVAS_FIXED_IMAGE` should be set to `FALSE` so that the server will inform the application, via the `CANVAS_RESIZE_PROC`, that the canvas has been resized. When this attribute is `TRUE`, the `CANVAS_RESIZE_PROC` is not called when the canvas is made smaller.
- If the application defines a color map for a canvas, the application must set the `WM_COLORMAP_WINDOWS` property on the its top-level window to tell the window manager which canvas will use the color map. The following lines of code shows how to accomplish this under XView.

```
Atom       catom;
/* These four lines get X info from XView objects */
display = (Display *)xv_get(frame, XV_DISPLAY);
```

```
frame_window = (Window)xv_get(frame, XV_XID);
canvas_window = (Window)xv_get
               (canvas_paint_window(canvas),XV_XID);
catom = XInternAtom(display, "WM_COLORMAP_WINDOWS", False);
XChangeProperty(display, frame_window, catom, XA_WINDOW, 32,
           PropModeAppend, &canvas_window, 1);
```

Note that an application does not need to set the `WM_COLORMAP_WINDOWS` property if it is only using one color map.

- Instead of calling `xgl_close()` after `xv_main_loop` returns, the application should call `xgl_close()` in a routine interposed on the quit event.

OLIT Tips

- The application should call `xgl_close()` from the “Quit” selection in the window menu. This is done by adding a callback to the toplevel shell:

```
OlAddCallback(toplevel, XtNwmProtocol, WMCallback, NULL);
```

The following lines of code show an example of a procedure that could be called when the shell receives a `WM_PROTOCOL` message.

```
void
WMCallback(widget, clientdata, calldata)
    Widget          widget;
    XtPointer       clientdata;
    XtPointer       calldata;
{
    OlWMProtocolVerify *wmvrfy = (OlWMProtocolVerify *)calldata;

    if ((wmvrfy->msgtype == OL_WM_DELETE_WINDOW) &&
                                (application wants to stay up))
        return;
    else {
        xgl_close(sys_state);
        OlWMProtocolAvction(widget, wmvrfy, OL_DEFAULTACTION);
    }
}
```

XGL Limitations

When you are writing XGL programs, note the following limitations:

- No Light objects are created at XGL initialization. This means that an XGL program does not have any lights automatically available. You can make lights available by creating a Light object or by setting the Context attributes `XGL_2D_CTX_LIGHT_NUM` and `XGL_3D_CTX_LIGHTS`. For more information on lighting, see "Lighting, Shading, and Depth Cueing" on page 395.
- Turning on error checking using the attribute `XGL_SYS_ST_ERROR_DETECTION` may affect performance. A good approach may be to set `XGL_SYS_ST_ERROR_DETECTION` to `TRUE` while you are developing your application and set it to `FALSE` when you compile your application for distribution. For more information on error checking, see "Error Detection and Reporting" on page 45.
- The concurrent use of backing store and double buffering is not supported. XGL enables whichever is requested first. For more information on backing store and double buffering, see page 63.
- Backing store (`XGL_WIN_RAS_BACKING_STORE`) should not be enabled when accumulation is needed or when `XGL_3D_CTX_HLHSR_MODE` is set to `XGL_HLHSR_Z_BUFFER` for a 3D Context. The reason for this is that the X server does not handle the copying of a Z-buffer or accumulation buffer for backing store. For more information, see page 64.
- Although you can mix XGL and Xlib calls, take care when mixing these calls because latencies involved with XGL communication with graphics hardware and Xlib communication with the server can result in geometry being drawn in an unexpected order. If the order in which the primitives are drawn is important to your application, you should take following precautions when mixing XGL and Xlib functions:
  - When the application has finished drawing with XGL primitives, call `xgl_context_flush(ctx,XGL_FLUSH_SYNCHRONIZE)` to ensure that XGL primitives are displayed before calling Xlib functions. This operator should be called even if the `XGL_CTX_DEFERRAL_MODE` is `XGL_DEFER_ASAP`.
  - Xlib calls should be flushed with `XSync` before further XGL commands are issued. However, the use of `XSync` may cause some performance degradation. Note also that when rendering and clearing the window with

Xlib calls, the application must clear the window with `xgl_context_new_frame()` before rendering with XGL primitives, even if the window has previously been cleared with Xlib calls.

- Your application must open XGL with a call to `xgl_open()` before it can call `xgl_inquire()`. Device characteristics cannot be queried before XGL is open. For information on `xgl_inquire()`. see page 86.
- Accumulation buffer functionality is currently only available on RGB rasters. For information on accumulation buffers, see page 222.
- Texturing is only available on 3D RGB rasters. For information on texture mapping, see "Texture Mapping" on page 451.

System State Information and Generic Operators

3

This chapter discusses the XGL System State object and provides information on the generic operators that can be used with any XGL object.

Introduction to the System State Object

An XGL application must begin with a call to the operator `xgl_open()`, which creates the XGL System State object. The XGL System State object maintains critical information that XGL requires for managing an XGL session. It keeps track of all other objects created by the application or by XGL itself. Only one XGL System State object within one XGL session is supported.

There are two System State operators, `xgl_open()` and `xgl_close()`, and several related System State attributes. These operators and attributes are explained in the following sections.

System State Operators

The operator `xgl_open()` creates and initializes a System State object.

```
Xgl_sys_state xgl_open (<attribute-list>);
```

If the call is successful, a handle to the System State object is returned to the application. As with many XGL operators, `xgl_open()` takes a `NULL`-terminated attribute list as one of its input parameters. It may have as few as

zero attribute-value pairs or as many as the total number of XGL system state attributes, but in all cases it must be terminated with a NULL character, where NULL is defined as in stdio.h. Each attribute-value pair consists of an attribute name followed by its value.

The operator xgl_close() ends an XGL session.

```
void xgl_close (Xgl_sys_state    system_state);
```

This operator is invoked using the handle to the XGL System State object previously created by the xgl_open() call. This operator destroys the XGL System State object and all its associated resources, and then terminates the XGL session.

System State Attributes

The application programmer can create a System State object and set its attributes simultaneously by specifying argument values in the xgl_open() call's list of attributes. Attributes of the System State object primarily affect error detection and error reporting. System State attributes also provide such information as the location of font data and the release number of the XGL library being used.

The attributes associated with the System State are:

XGL_SYS_ST_ERROR_DETECTION
: This attribute enables or disables additional error detection and reporting. The attribute can either be TRUE or FALSE. TRUE indicates that extra error checking will be done for errors in the USER, ARITHMETIC, and CONFIGURATION error categories (at the expense of performance). FALSE (the default value) limits error checking and reporting to RESOURCE and SYSTEM errors. For information on the XGL error categories, see Table 3-1 on page 47.

XGL_SYS_ST_ERROR_NOTIFICATION_FUNCTION
: This attribute enables an application to supply its own error-reporting function in place of the XGL default error function. The default value is a pointer to a function that prints an error message to the standard error output (stderr).

XGL_SYS_ST_ERROR_INFO
This attribute is used to get information about the current error state. It can be used within an application's error notification function to change the function's behavior based on specific error characteristics.

XGL_SYS_ST_VERSION
This read-only attribute queries the release number of the current library.

XGL_SYS_ST_SFONT_DIRECTORY
This attribute sets the path used by the XGL library to access stroke font data files used in the stroke font primitives. When the XGL library is initialized, the System State object sets the stroke font path to $XGLHOME/lib/xglfonts/stroke if the XGLHOME environment variable is set or to /opt/SUNWits/Graphics-sw/xgl/lib/xglfonts/stroke if XGLHOME is not set. It then stores the path name in the attribute XGL_SYS_ST_SFONT_DIRECTORY.

The application can store the stroke fonts in a directory other than the directory in which they were installed. The XGL_SYS_ST_SFONT_DIRECTORY attribute enables the application to set the stroke font directory path. If the stroke fonts are stored in a location other than the default location, this attribute must be set every time the application is restarted. If XGL does not find the stroke font data when rendering stroke text, an error is reported, and the stroke rendering operation stops.

Error Detection and Reporting

When an XGL application causes an error, the error notification function determines the application-visible response. Because this function is settable by the application, its response can vary depending on its current definition. The default error notification function sends an error message to stderr. For example, the following message is produced by the default function for a malloc error that occurs within an xgl_polygon() call from a 3D Context:

```
Error number di-1: malloc or new failed:  out of memory
Operator: xgl_polygon
Object: XGL_3D_CTX
```

Error Notification Function

The XGL library provides the default error notification function. When an error occurs, this function sends an ASCII string to the standard error output (`stderr`). The string consists of the error code associated with the error, a message describing the error, the XGL operator being executed when the error was detected, the XGL object being used, and other optional information.

XGL allows the application to supply its own error function for filtering errors and reporting them to the application program. An application error notification function can specify the error categories of interest to the application and ignore other error categories. The following code fragment shows an example of an application error notification function that ignores error messages caused by singular transform errors.

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

/*   error_notify
 *      Error handling code to prevent singular transforms from
 *      issuing error messages. All other XGL errors are displayed
 *      as if this procedure did not exist.
 */

static Xgl_sgn32 error_notify(Xgl_sys_state sys_state)
{
    Xgl_error_info info;       /* Information structure */

    xgl_object_get(sys_state, XGL_SYS_ST_ERROR_INFO, &info);

    if (strcmp("di-128", info.id) == 0) {
        /* Singular transform, we did it on purpose, ignore */
        return(0);
    }

    /* Just print out the standard error messages */
    if (info.msg != NULL)
        printf("msg = %s\n", info.msg);
    if ((info.cur_op != NULL) && (info.cur_obj != NULL))
        printf("cur_op = %s\t%s\n", info.cur_op,info.cur_obj);
    if ((info.operand1 != NULL) && (info.operand2 != NULL))
        printf("cur_obj = %s\t%s\n", info.operand1, info.operand2);

    return(1);
}
```

This function uses the information in the *Xgl_error_info* structure returned by XGL_SYS_ST_ERROR_INFO to handle errors. Any of the eight fields in the structure can be used to filter errors. The System State attribute XGL_SYS_ST_ERROR_NOTIFICATION_FUNCTION sets the application-specific error function, as in this example:

```
/* Prepare to catch error 128, singular transform */
   xgl_object_set(sys_st,
        XGL_SYS_ST_ERROR_NOTIFICATION_FUNCTION, error_notify,
        NULL);
```

The interface for the error notification function is:

```
error_notify (Xgl_sys_state sys_state);
```

Error Types and Categories

XGL errors are grouped into five categories, as listed in Table 3-1. These error categories enable an application programmer creating a new error notification function to detect a particular error group and respond accordingly.

Table 3-1 Error Categories

Category	Description
SYSTEM	Internal errors, unsupported features, and errors that generally cannot be fixed by changing the application.
CONFIGURATION	Errors caused by improper installation or configuration of XGL (such as a .so file not found).
RESOURCE	Unavailable resource errors including both hardware and software resources (such as memory, shared memory, window ID, frame buffer).
ARITHMETIC	Arithmetic exceptions (such as an error resulting from dividing by 0 or taking the square root of -1).
USER	Errors caused by invalid function parameters, nonexistent user files, or situations that may be caused by application program logic errors.

Errors in these categories can be further classified into RECOVERABLE and NONRECOVERABLE error types, as described in Table 3-2. Recoverable errors are reported with error codes of 100 or greater. With recoverable errors, XGL makes an assumption about what the application intended, corrects the error, and continues. If an error is nonrecoverable, XGL stops processing and returns control to the caller. Nonrecoverable errors may cause a core dump with further execution of the application because the application will be left in an invalid state. The application should stop further processing if XGL returns with a nonrecoverable error. Nonrecoverable errors are reported with error codes between 0 and 99.

Table 3-2 Error Types

Type	Description
NONRECOVERABLE	XGL immediately aborts processing and returns control to the caller. Includes all SYSTEM and CONFIGURATION errors, most RESOURCE errors, and some ARITHMETIC errors.
RECOVERABLE	XGL makes assumptions about what the application intended to do, corrects the error if it can, and continues processing. Includes some RESOURCE errors, most ARITHMETIC errors, all USER errors.

System errors generally cannot be fixed by altering the application program. Other kinds of errors are caused by improper use of the XGL library and can be fixed by modifying the application. The following example error notification function checks the error ID; this function could be extended to stop processing in the case of a nonrecoverable error.

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

static Xgl_sgn32 error_notify2(Xgl_sys_state sys_state)
{
   Xgl_error_info     info;
   int                n;

   xgl_object_get (sys_state, XGL_SYS_ST_ERROR_INFO, &info);

   printf ("id      = %s\n", info.id);
   printf ("msg     = %s\n", info.msg);
   printf ("cur_op  = %s\n", info.cur_op);
```

```
    printf ("cur_obj = %s\n", info.cur_obj);

    n = atoi(info.id);
    if (n < 100) {
         printf ("Non-Recoverable Error!\n");
    } else {
         printf ("Recoverable Error!\n");
    }

    return(1);
}
```

Note that the XGL default error notification function does not use the error types and categories. These are provided for developers who want to trap specific error types and categories in their own error notification function.

Note – When error checking is turned on (`XGL_SYS_ST_ERROR_DETECTION` is set to `TRUE`), XGL checks as many errors as possible. This may affect performance. Therefore, you should set `XGL_SYS_ST_ERROR_DETECTION` to `TRUE` while you are developing your application and set it to `FALSE` when you compile your application for distribution.

Generic XGL Operators

The XGL library provides four operators that are used to manipulate all XGL objects. These operators provide a consistent method of working with XGL objects. XGL objects no longer needed by an application should be destroyed so the system can reclaim resources (mainly memory) for use by new objects. Destroying unused objects minimizes the amount of internal data structures, thereby improving performance when referencing objects.

Note – The create and destroy operators described below cannot be used on the System State object. The `xgl_open()` and `xgl_close()` operators create and destroy the System State object.

The generic XGL operators are:

- `xgl_object_create()`
  This operator creates the XGL object specified by *type* and initializes it with the settings provided in the *attributes* list. After creating the object, the application can determine the values of the object's attributes using the `xgl_object_get()` operator.

```
Xgl_object  xgl_object_create (
    Xgl_sys_state       sys_state,
    Xgl_obj_type        type,
    Xgl_desc            *desc,
    <attribute-list>    attributes)
```

The parameter *type* can take the values listed in Table 3-3.

Table 3-3 Values of the `xgl_object_create` *type* Parameter

Value	Creates:
XGL_2D_CTX	2D Context object
XGL_3D_CTX	3D Context object.
XGL_CGM_DEV	XGL CGM Device object
XGL_CMAP	Color Map object
XGL_DMAP_TEXTURE	Data Map Texture object
XGL_GCACHE	Gcache object
XGL_LIGHT	Light object
XGL_LPAT	Line Pattern object
XGL_MARKER	User-defined Marker object
XGL_MEM_RAS	Memory Raster object
XGL_MIPMAP_TEXTURE	MipMap Texture object
XGL_PCACHE	Pcache object
XGL_SFONT	Stroke Font object
XGL_STREAM	Stream Device object
XGL_SYS_STATE	System State object

Table 3-3 Values of the `xgl_object_create` *type* Parameter *(Continued)*

Value	Creates:
XGL_TMAP	Texture Map object
XGL_TRANS	Transform object
XGL_WIN_RAS	Window Raster object

The parameter `desc` is used to define additional information needed for the Stroke Font object, the Window Raster Device object, and the Stream Device object. In the case of the Stroke Font object, the `sfont_name` field of the `desc` parameter contains a pointer to the name of the font. For the Window Raster Device object, the `desc` parameter contains a window type and a window descriptor whose contents depend on the window type. For a Stream Device object, the `stream` structure of the `desc` parameter consists of a field pointing to the name of the specific stream pipeline and a field containing a pointer to device-specific information for the particular stream device.

- `xgl_object_destroy()`
  This operator destroys the specified XGL object (represented by the handle returned by the corresponding `xgl_object_create()` call) by freeing all the resources associated with it. The actual destruction of an object may be delayed if the object is referenced by other objects.

```
void xgl_object_destroy (Xgl_object   handle)
```

- `xgl_object_get()`
  This operator is used to determine the value of attributes associated with an object. It returns the value of a specified attribute, *attr*, of the object, *obj*. The parameter *val* points to a memory location into which the operator writes the attribute's value.

```
void xgl_object_get (
    Xgl_object     obj,
    Xgl_attribute  attr,
    void           *val)
```

For example, the following call to `xgl_object_get()` retrieves the value of the current View Transform and stores it in `view_trans`:

```
xgl_object_get(ctx, XGL_CTX_VIEW_TRANS, &view_trans);
```

Note – The application program is responsible for ensuring that the *val* field in the `xgl_object_get()` operator points to an area in memory large enough to store the information returned.

- `xgl_object_set()`
  This operator is used to change the value of attributes associated with an object. It assigns a given value to an attribute in the object *obj*. One or more attributes can be updated with a single call to `xgl_object_set()`. The parameter type <*attribute-list*> is used to specify a `NULL`-terminated list of attribute-value pairs. For each pair, the attribute name is followed by the value.

```
void xgl_object_set (
    Xgl_object          obj,
    <attribute-list>    attributes)
```

For example, the following call to `xgl_object_set()` sets four Context edge pattern attributes:

```
xgl_object_set (ctx,
            XGL_CTX_EDGE_PATTERN,      xgl_lpat_dashed_dotted,
            XGL_CTX_EDGE_COLOR,        &ln_fg_color,
            XGL_CTX_EDGE_ALT_COLOR,    &ln_bg_color,
            XGL_CTX_SURF_FRONT_COLOR, &pgon_front_color,
            NULL);
```

Devices 4

This chapter discusses the Device objects. It includes information on the following topics:

- Functionality and characteristics of the Device object.
- Examples of XGL programs using Xlib, XView, and OLIT.
- Determining hardware acceleration features from within an application program.

Introduction to Device Objects

XGL defines a *device* as any surface on which it can render (create) an image, whether the image is visible or invisible. The Device class is an abstraction of many types of graphics display devices. It has two subclasses: *Raster*, which represents a two-dimensional rectangular array of discrete image samples (pixels), and *Stream*, which represents any picture representation, such as a Computer Graphics Metafile (CGM), that doesn't rely on discrete pixels to display the image.

The pixels in a raster can reside in a graphics frame buffer (that is, on the display device) or in memory. Thus, there are two types of Raster objects: *Memory Rasters*, which correspond to contiguous areas of memory that XGL will write into, and *Window Rasters*, which are areas on the display screen, managed by the window system, that XGL will write into.

A Window Raster requires a window system in which to run; it does not work with a raw screen. The actual display device containing the pixels is generally a window canvas created by the X11 server running as part of the window system. The application must execute the proper Xlib or X toolkit calls to create this canvas before attaching an XGL Window Raster Device. A window should have only one associated Window Raster Device object attached to it.

A Memory Raster is a block of memory allocated from main memory. XGL draws into it the same way it draws into Window Rasters. The pixels composing a Memory Raster Device are in the application's memory space.

XGL provides one Stream device at this time, a CGM device. A CGM device outputs CGM formatted graphics information to the output stream.

The graphics hardware devices running applications may differ considerably. Because XGL Device objects provide the framework for interacting with graphics devices in an abstract, device-independent manner, in most cases the application programmer does not need to know the hardware specifics of each system.

Creating Device Objects

Device objects are created with the `xgl_object_create()` operator using the input values in Table 4-1:

Table 4-1 Device Object Types

Device Type	*type* Parameter	*desc* Parameter
Window Raster	`XGL_WIN_RAS`	A pointer to an *Xgl_obj_desc* structure containing a window type and a window descriptor.
Memory Raster	`XGL_MEM_RAS`	`NULL`
Stream	`XGL_STREAM`	A pointer to an *Xgl_obj_desc* structure containing a pointer to the name of the library for the stream device and a pointer to any device-specific information for the stream device.
CGM	`XGL_CGM_DEV`	A pointer to an *Xgl_obj_desc* structure containing a pointer to the name of the library for the XGL-provided CGM Stream device.

Window Raster Device Object

To create a Window Raster Device object, XGL needs information on the X11 window that XGL will render to. This information is provided in the *desc* parameter, which points to a *Xgl_obj_desc* union containing a window type and a window descriptor structure. The only supported value for the window raster type is XGL_WIN_X. The window descriptor structure is:

```
typedef struct {
    void        *X_display;
    int          X_screen;
    Xgl_usgn32   X_window;
} Xgl_X_window;
```

The window system type can be modified to indicate which protocol XGL uses to communicate with the graphics device. XGL supports three protocols for the X window system: Xlib, PEX, and DGA (Direct Graphics Access). The protocol is selected by OR'ing in the name of the protocol with the constant XGL_WIN_X. The constants for the names of the protocol are:

```
#define XGL_WIN_X_PROTO_DEFAULT        (0x000)
#define XGL_WIN_X_PROTO_XLIB           (0x100)
#define XGL_WIN_X_PROTO_PEX            (0x200)
#define XGL_WIN_X_PROTO_DGA            (0x400)
```

Only one protocol can be specified. If a protocol is specified, XGL tries to use that protocol to communicate with the graphics device. If it is not available, an error is issued and the Window Raster is not created. If no protocol is specified (or XGL_WIN_X_PROTO_DEFAULT is used), XGL tries to use the best protocol for the particular device. If that is not available, XGL will try a backup protocol. In practical terms, XGL tries to use DGA, but if that is not available, it will attempt to use PEX, and if that fails, then it will use Xlib.

In most cases, an XGL application does not need to be concerned with selecting a protocol. However, this facility is available to aid applications that want to make the best use of the graphics resources, and this usually means part of the application that is device-dependent.

The code fragment below shows how a Window Raster object can be created in XView.

```
Xgl_win_ras       win_ras = NULL;/* Xgl window raster */
Xgl_X_window      xgl_x_win;     /* XGL-X data structure */
Xgl_obj_desc      win_desc;      /* XGL window raster structure */

pw = (Xv_Window) canvas_paint_window(canvas);
canvas_window = (Window) xv_get(pw, XV_XID);
frame_window = (Window) xv_get(frame, XV_XID);

/* put X stuff into XGL data structure */
xgl_x_win.X_display = (void *) XV_DISPLAY_FROM_WINDOW(pw);
xgl_x_win.X_window = (Xgl_usgn32) canvas_window;
xgl_x_win.X_screen = (int) DefaultScreen(display);

/* create Window Raster Device using XView canvas */
win_desc.win_ras.type = XGL_WIN_X;
win_desc.win_ras.desc = &xgl_x_win;
win_ras = xgl_object_create(sys_st, XGL_WIN_RAS, &win_desc,
                              NULL);
```

A Device object must be associated with a Context object for rendering to occur. The two objects are associated using the XGL_CTX_DEVICE attribute. This association can be made when the Context object is created, as in the following code fragment.

```
ctx = xgl_object_create(sys_st, XGL_2D_CTX, NULL,
                   XGL_CTX_DEVICE, win_ras,
                   NULL);
```

The association between the Device and the Context can also be set with the `xgl_object_set()` operator:

```
xgl_object_set(ctx,XGL_CTX_DEVICE, win_ras,NULL);
```

Note – The current implementation of XGL, using DGA, does not support `fork()` or `vfork()` if there is an active XGL Window Raster. The application can create only one XGL Raster for each OpenWindows (X) window.

Stream Device Object

The XGL Stream device provides protocol-independent stream pipelines for creating formatted output such as CGM output. The interface to the Stream device is independent of each specific Stream implementation. Each implemented Stream device provides an `include` file that contains the mapping of from the pipeline-specific attributes to the generic Stream attributes.

To create a Stream device object, the application needs to supply XGL with the name of the library that provides the Stream device functionality, and, if the Stream pipeline requires other information, a pointer to that information. This information is provided to XGL in the `xgl_object_create()` *desc* parameter, which points to an ***Xgl_obj_desc*** union containing a `stream` structure. The actual name of the library is defined in the header file provided with the device pipeline for the Stream device. See the documentation for the particular Stream device for specific information on the name of the library and for required device-specific information. For an example of a Stream device, see “CGM Device Object” on page 66.

A Stream device provides support for the following XGL attributes:

- `XGL_OBJ_APPLICATION_DATA`
- `XGL_OBJ_TYPE`
- `XGL_OBJ_SYS_STATE`
- `XGL_DEV_COLOR_MAP`
- `XGL_DEV_REAL_COLOR_TYPE`
- `XGL_DEV_COLOR_TYPE`
- `XGL_DEV_CONTEXTS`
- `XGL_DEV_CONTEXTS_NUM`
- `XGL_DEV_MAXIMUM_COORDINATES`

The Stream device may provide additional attributes that are passed through XGL from the application to the pipeline and from the pipeline to the application. After creating a Stream device, the application can inquire the values of the attributes using the `xgl_object_get()` operator. See the documentation for the Stream device for information on available attributes.

4

Raster Object Attributes

Raster attributes define the Window and Memory Raster Device objects created by the Device object operators. Some are shared by both Memory and Window Device objects, and others are specific to one or the other.

General Raster Attributes

Raster attributes that affect all Raster objects are:

`XGL_DEV_COLOR_MAP`

This attribute allows the application to establish its own color map in the Raster Device for specifying the colors rendered onto the Raster. Color maps apply to Rasters of type `XGL_COLOR_INDEX`. They are not applicable to Rasters of type `XGL_COLOR_RGB` since user colors are mapped directly to hardware lookup tables. Once a Color Map object has been created, it can be attached to the Raster using `xgl_object_set()` and this attribute. The default value is a handle to a Color Map object containing a two-element, black and white color table.

After an application sets this attribute and before attempting to draw into the Raster, the application must call `xgl_context_new_frame()` on the Context to which the Raster is attached. This ensures the correct colors.

`XGL_DEV_REAL_COLOR_TYPE`

This is a read-only attribute that indicates the actual color space used to store pixels in an XGL Device and the information contained in an actual pixel. This information is device-dependent.

`XGL_DEV_COLOR_TYPE`

This attribute defines the color space used for the calculation of color and lighting. It is set at the time of Raster creation and can only be set once. The color space of the device for a window raster object is directly related to the visual class of the X window underlying the window raster object. If the visual class of the X window is PseudoColor, then the color type of the window raster object should be `XGL_COLOR_INDEX`. If the visual class of the X window is TrueColor or DirectColor, the color type of the window raster object should be `XGL_COLOR_RGB`**.** The color spaces currently supported in XGL as defined by the data type *Xgl_color_type* are:

- XGL_COLOR_RGB, where color is specified as a fractional component of each of the three primary colors: red, green, and blue. Each component is a floating-point number between 0.0 and 1.0, which represents the additive weight the primary contributes to the final color. The visual type of the X window underlying a window raster of this color type should be TrueColor or DirectColor. Note that if the visual type is DirectColor, XGL will behave as if on a TrueColor visual.
- XGL_COLOR_INDEX, where color is specified as an unsigned integer index into a color table. The color table of an XGL Color Map object maps index colors into RGB color values. The default value is device-dependent. The visual type of the X window underlying a window raster of this color type should be PseudoColor.

Thus, for example, if the visual class of the X window underlying the window raster object is PseudoColor, the color type of the Window Raster object is set as shown in this code fragment:

```
if (vis->class == PseudoColor)
    ras = xgl_object_create(XGL_WIN_X, &xgl_x_win,
                    XGL_DEV_COLOR_TYPE, XGL_COLOR_INDEX, NULL);
```

If the visual class of the X window is TrueColor or DirectColor, the color type of the window raster object is set to XGL_COLOR_RGB as follows:

```
if (vis->class == TrueColor || vis->class == DirectColor)
    ras = xgl_object_create(XGL_WIN_X, &xgl_x_win,
                    XGL_DEV_COLOR_TYPE, XGL_COLOR_RGB, NULL);
```

Note – For good performance, you should understand the color model of the target device and program accordingly. Take care when selecting the visual because the chosen visual class affects the XGL hardware color type, which in turn affects performance. For example code showing the proper way to select the visual that gives the best performance on any frame buffer, see the *XGL Accelerator Guide for Reference Frame Buffers.*

XGL_DEV_CONTEXTS_NUM

This read-only attribute returns the number of Context objects associated with a Device. The default value is 0.

XGL_DEV_CONTEXTS
This read-only attribute returns a list of the Context object handles associated with a Device. To use this attribute, the application should first get the number of Contexts associated with the Device using the XGL_DEV_CONTEXTS_NUM attribute, allocate sufficient memory to hold the array of object handles, where each object handle is *Xgl_ctx* in size, and then get the list of object handles. The default value is NULL.

XGL_DEV_MAXIMUM_COORDINATES
This read-only attribute returns the Device's maximum *x* and *y* coordinates values. In addition, the maximum Z-buffer value allowed by the Device, is returned in the *z* value of the return *Xgl_pt_f3d* argument. The default values are device-dependent.

XGL_RAS_DEPTH
This attribute defines the number of bits used to specify the color of one pixel in an XGL Raster. It is read-only for Window Rasters and read-write for Memory Rasters. Four depths are supported: 1, 4, 8, and 32. The 1-bit depth is used only for creating stipple patterns by the xgl_context_copy_buffer() operator and other polygon-fill attributes. The 4-bit, 8-bit, and 32-bit depths are used for Memory Rasters into which the XGL application can render. The default value is 8 for Memory Rasters. If the depth of the Memory Raster is 8, the recommended device color type is XGL_COLOR_INDEX. If the depth of the Memory Raster is 32, the only device color type that is supported is XGL_COLOR_RGB. The default value is device-dependent for Window Rasters.

XGL_RAS_HEIGHT
This attribute defines the height of an XGL Raster. Window Raster height is obtained from the size of the window passed to the object create call. Memory Raster height can be set or reset at any time. The maximum Memory Raster height is 4K. Changing the height of a Memory Raster results in the loss of all pixel data it contains at that time. The default value is 256 for a Memory Raster.

XGL_RAS_WIDTH
This attribute defines the width of an XGL Raster. The Window Raster width is obtained from the size of the window passed to the object create call. The Memory Raster width can be set or reset at any time. The maximum Memory Raster width is 4K. Changing the width of a Memory Raster results in the loss of all pixel data it contains. The default value is 256 for a Memory Raster.

XGL_RAS_SOURCE_BUFFER
This attribute specifies the source buffer for the raster operations `xgl_context_copy_buffer()` and `xgl_context_get_pixel()`. The source buffer can be a raster's draw buffer or its Z-buffer. The default value is the draw buffer.

XGL_RAS_RECT_NUM
This attribute specifies the number of clip rectangles in the application clip list.

XGL_RAS_RECT_LIST
This attribute specifies the list of clip rectangles in the application clip list.

Window Raster Attributes

Raster attributes that affect all Window Raster objects are:

XGL_WIN_RAS_TYPE
This attribute supplies the type of Window Raster Device. It is a read-only attribute that describes the window system controlling the Window Raster. The value of this attribute is the value the application passes to XGL with the operator `xgl_object_create()`. XGL returns the following value:

XGL_WIN_X The Window Raster is an X11 window.

XGL_WIN_RAS_DESCRIPTOR
This attribute is the *descriptor* passed to XGL on the creation of the Window Raster. Its value is the same as the value passed to XGL by the operator `xgl_object_create()`. It returns a value of type *Xgl_X_window*, as X11 windows are the only supported window type. There is no default value.

XGL_WIN_RAS_POSITION
This read-only attribute specifies the position of the Window Raster object. It returns the position of the window in device coordinates relative to the upper left corner of the display screen.

XGL_WIN_RAS_PIXEL_MAPPING
This attribute specifies the pixel mapping array from the user's index colors to the window system color map index colors. This attribute must be set by the application immediately after setting XGL_CMAP_NAME for the Color

Map object associated with the Window Raster. The returned pixel array of the color map created for the Window System is sent to XGL via XGL_WIN_RAS_PIXEL_MAPPING. The default value is *identity*.

XGL_WIN_RAS_STEREO_MODE
This attribute specifies whether the window being rendered into is a stereo window or a monocular (non-stereo) window. If the attribute is set to XGL_STEREO_NONE and the frame buffer is in stereo mode, all graphics will be rendered into both the left and right eye buffers. If the value is set to XGL_STEREO_LEFT, the image for the left eye buffer is rendered; if the value is set to XGL_STEREO_RIGHT, the image for the right eye buffer is rendered. Note that undefined results may occur if stereo is used when rendering through a 2D Context.

XGL_WIN_RAS_BUF_DISPLAY
This attribute specifies the current display buffer in the application window. When the display and draw buffers are equivalent, the Window Raster is in single-buffer mode. The default value is 0, and the default buffering value is single buffer.

XGL_WIN_RAS_BUF_DRAW
This attribute specifies the buffer for reading and writing pixels. All XGL drawing primitives take place in this buffer. The default value is 0.

XGL_WIN_RAS_BUF_MIN_DELAY
This attribute specifies the minimum number of milliseconds to wait between switches of the display and draw buffers. Minimum delay allows the application to control the speed of the animation loop so it appears synchronized at a rate the application can support. The default value is 0.

XGL_WIN_RAS_BUFFERS_REQUESTED
This attribute allocates a specified number of hardware buffers if supported by the hardware. If the hardware does not support multi-buffering, then only 1 buffer will be allocated. If the application sets the number to 1, XGL will do single buffering. If the value is set to 2, XGL will do double buffering, if double buffering is supported by the hardware and the underlying Window Raster. The application should perform a *get* on the attribute XGL_WIN_RAS_BUFFERS_ALLOCATED immediately after setting XGL_WIN_RAS_BUFFERS_REQUESTED, to determine how many buffers were allocated in the hardware. If the hardware supports only a single buffer, color map double buffering is possible. (For information about Color Map double buffering, see Chapter 5, “Color”.)

The value of this attribute determines the valid values for the attributes XGL_WIN_RAS_BUF_DISPLAY and XGL_WIN_RAS_BUF_DRAW. The default value is 1 (single buffered).

XGL_WIN_RAS_BUFFERS_ALLOCATED
This read-only attribute reflects the number of buffers allocated in the hardware underlying a Window Raster. The application must first set the XGL_WIN_RAS_BUFFERS_REQUESTED attribute to either 1 (single buffered) or 2 (double buffered). The value of this attribute determines the valid values for the attributes XGL_WIN_RAS_BUF_DISPLAY and XGL_WIN_RAS_BUF_DRAW. The default value is 1 (single buffered).

Note – Since the concurrent use of backing store and double buffering is not supported, XGL will enable whichever is requested first. If a request for double buffering is made by setting XGL_WIN_RAS_BUFFERS_REQUESTED to 2 and backing store has already been enabled, XGL will not initiate the double buffering request and will set the value of XGL_WIN_RAS_BUFFERS_ALLOCATED to 1.

XGL_WIN_RAS_BACKING_STORE
This attribute enables backing store support for a Window Raster, after backing store support is requested through the Xlib call XChangeWindowAttributes(). Since concurrent use of backing store and double buffering is not supported, XGL will enable whichever is requested first. If a request for backing store is made by setting XGL_WIN_RAS_BACKING_STORE to TRUE and the value of XGL_WIN_RAS_BUFFERS_ALLOCATED is greater than 1, XGL will not initiate the backing store request and will set the value of XGL_WIN_RAS_BACKING_STORE to FALSE.

Note – The X server does not handle the copying of a Z-buffer or accumulation buffer for backing store. When a Z-buffer and/or an accumulation buffer is enabled and the display device uses a software Z-buffer and/or accumulation buffer, the backing store will share the software buffers with the display window, so the contents of the buffers will be the same without copying. If the display device uses a hardware Z-buffer and/or accumulation buffer, the buffers cannot be shared with the backing store. In this case, the device pipeline for the backing store will use a separate Z-buffer for rendering and/or a separate accumulation buffer to accumulate. The resulting image after damage repair is likely to be inaccurate, especially if the clip list has changed. Thus, backing store should not be used when accumulation is needed or when `XGL_3D_CTX_HLHSR_MODE` is set to `XGL_HLHSR_Z_BUFFER` for a 3D Context.

Memory Raster Attributes

`XGL_MEM_RAS_IMAGE_BUFFER_ADDR`

This attribute reflects the address of the array of pixels for an XGL Memory Raster, allowing the application programmer to read and write pixel data. The application programmer must know the depth, width, and height of a Raster to access its pixel values correctly. The memory address varies depending on the Raster depth:

- 1-bit Rasters: The memory address points to the word containing the upper left pixel. The pixels are stored along the row as bits, the left-most pixel being the most significant bit (MSB) in the word. To allow each row to begin at a 16-bit boundary, padding can be added to the least significant bits of the word at the end of each row.
- 4-bit Rasters: The memory address points to the word containing the upper left pixel. Each byte contains two pixels stored along the row. Each row begins at a 16-bit boundary, and padding may be added to the row.
- 8-bit Rasters: The memory address points to the word containing the upper left pixel. Each byte contains one pixel, stored as an unsigned char, along the row. Each row begins at a 16-bit boundary.
- 32-bit Rasters: The memory address points to the word containing the upper-left pixel. Each pixel occupies 32 bits, and the pixels are aligned to the 32-bit boundary.

Note – When creating a Memory Raster, the application program must set the XGL_RAS_DEPTH, XGL_RAS_WIDTH, and XGL_RAS_HEIGHT attributes in the xgl_object_create() call before setting the XGL_MEM_RAS_IMAGE_BUFFER_ADDR attribute.

XGL_MEM_RAS_Z_BUFFER_ADDR

This attribute reflects the starting address of the block of memory for the Z-buffer of a Memory Raster, allowing the application programmer to read and write Z-buffer values.

The Z-buffer is a collection of *z* values. Each *z* value occupies 32 bits. XGL accesses the least significant three bytes of the 32-bit word during Z-buffer operations. Thus, each row begins at a 32-bit word boundary. The Z-buffer of a Memory Raster has 24 bit planes and thus a depth of (2^{24} - 1). The height of the Z-buffer is based on the raster height (XGL_RAS_HEIGHT), and the width is based on the raster width (XGL_RAS_WIDTH).

Note – When copying from an indexed 8-bit Memory Raster into an indexed 24-bit Window Raster, a separate Color Map object is required for each raster. Since XGL does internal color map conversion, which is different in these cases, the same color map cannot be used when the underlying hardware is different.

Note – If your application copies from a memory raster to a window raster and the color map being used does not start at zero, you may not get the colors you expect. The ROP mode for this copy buffer case sets the start index value for the color map to zero. To solve this problem, initialize the area using the xgl_context_new_frame() operator.

4

CGM Device Object

XGL can open and write to a Computer Graphics Metafile (CGM), Version 1. The XGL CGM file conforms to the ISO 8632:1985 standard. Metafiles are UNIX file streams that contain graphical information in a special format, and are used to store or interchange graphical information. A CGM is not a Raster Device. It is a data file consisting of a stream of graphical information. The CGM object uses higher-level primitives; the resulting output is compact and often easier to manipulate than a raster file. For example, modifying orientation is easier on CGM polygon data than on a polygon-shaped pixel area of a raster file.

The XGL CGM Device supports both 2D and 3D Contexts. However, the CGM standard deals strictly with 2D coordinates. The XGL implementation is further restricted to outputting only 2D integer coordinates. Although the CGM Device outputs only *x* and *y* coordinates, all stages of the XGL viewing pipeline are still supported. In the current implementation, stroke text, 3D circles, circular arcs, ellipses, elliptical arcs, and NURBs are output as polylines or polygons. Lines are interpolated, and surfaces are constant shaded. Picking is not supported. Although point colors and normals are ignored by the CGM device, the XGL pipeline processes them; therefore, face culling and face distinguishing are possible on a CGM device. Pixel operations are not supported in the current implementation. User-defined line and fill patterns are not supported.

The XGL implementation supports Binary, Character, and Clear Text CGM encoding formats. Both Long and Short integer coordinate representations are available in the current implementation. Floating point coordinates are not currently supported.

Creating a CGM Device

To create a CGM device, the application must supply the name of the XGL CGM library, `XGL_CGM_LIB_NAME`, in the Stream object descriptor. The name of the library is defined in the header file provided with the CGM device. At device creation time, the application can also pass in Stream device attributes. The code fragment below shows the creation of an XGL CGM device.

```
#include <stdlib.h>
#include <X11/Xlib.h>
#include <xgl/xgl.h>
#include <xgl/xgl_cgm-2.0.h>
```

```
static Xgl_3d_ctx     ctx = NULL;/* XGL context object */
static Xgl_sys_state  sys_st;    /* XGL system state object */

main (argc, argv)
   int     argc;
   char    *argv[];
{
    Display           *display;   /* pointer to X display */
    Xgl_obj_desc      obj_desc;   /* XGL object descriptor */
    char*                 fname = "CGM_file";
    float                 scale_factor = 1000.0;
    Xgl_cgm               cgm;
    Xgl_cgm_description   cgm_desc;

    if (argc == 2)
        fname = argv[1];

    sys_st = xgl_open(NULL);

    /* Use the XGL CGM pipeline provided with Solaris */
    obj_desc.stream.name = XGL_CGM_LIB_NAME;
    obj_desc.stream.desc = NULL;

    cgm_desc.cgm_fp = fopen(fname, "w");
    cgm_desc.cgm_signature = NULL;
    cgm_desc.cgm_description = NULL;
    cgm = xgl_object_create(sys_st,
                XGL_CGM_DEV, &obj_desc,
                XGL_DEV_COLOR_TYPE, XGL_COLOR_INDEX,
                XGL_CGM_DESCRIPTION, &cgm_desc,
                XGL_CGM_ENCODING, XGL_CGM_CLEAR_TEXT,
                XGL_CGM_VDC_EXTENT, XGL_CGM_VDC_EXT_SHORT,
                XGL_CGM_SCALE_MODE, XGL_CGM_METRIC,
                XGL_CGM_SCALE_FACTOR, &scale_factor,
                NULL);

  /* create XGL Context object using the CGM device */
  ctx = xgl_object_create (sys_st,
                XGL_3D_CTX, NULL,
                XGL_CTX_DEVICE, cgm,
                XGL_CTX_DEFERRAL_MODE, XGL_DEFER_ASAP,
                XGL_CTX_NEW_FRAME_ACTION, XGL_CTX_NEW_FRAME_CLEAR,
               NULL);
}
```

Creating a Picture

The `xgl_object_create()` operator results in the following lines being inserted into a metafile on a CGM Device outputting clear text:

```
(open file)
BEGIN METAFILE <string1> /*signature from description*/
METAFILE VERSION 1
METAFILE DESCRIPTION <string2>
VDC TYPE INTEGER /*all metafiles are this type*/
METAFILE ELEMENT LIST
DRAWING BEGMFDEFAULTS ENDMFDEFAULTS COLRINDEXPREC MAXCOLRINDEX
INTEGERPREC REALPREC INDEXPREC COLORPREC COLRVALUEEXT
SCALEMODE COLRMODE LINEWIDTHMODE MARKERSIZEMODE
```

The `xgl_object_destroy()` operator destroys the CGM Device. When `xgl_object_destroy()` is invoked, the CGM device closes the current picture, if it isn't already closed. The CGM Device then ensures that an "end metafile" delimiter has been written, and flushes the output stream. The user must close the stream. As with any XGL device, a call to `xgl_object_destroy()` will not have an effect until all XGL objects using the device are destroyed. Overlooking this can result in incomplete metafiles, since the file will not be flushed or closed.

Invoking the `xgl_context_new_frame()` operator (used to delimit CGM pictures) starts a new picture element in the Metafile. The current color map, which should be loaded by this time, is output to the Metafile. The `xgl_context_new_frame()` operator results in the output of the "begin picture" delimiter to the Metafile. Any picture already open is closed, and a new picture is started.

The following Context attributes apply to the CGM Device:

```
XGL_CTX_DEFERRAL_MODE
XGL_CTX_DEVICE
XGL_CTX_VDC_ORIENTATION
XGL_CTX_BACKGROUND_COLOR
```

CGM Metafile Device Attributes

Since a CGM Device is not a Raster Device, none of the Raster Device-specific attributes apply. Attributes that affect CGM Device objects are:

`XGL_CGM_TYPE`

This read-only attribute defines the type of CGM Metafile object associated with an XGL object; it cannot be modified later. It can be used to query the CGM Metafile object type when only a generic *Xgl_cgm* handle is known. The only type of CGM Metafile object that the XGL library currently supports is `XGL_CGM_V1`.

`XGL_CGM_ENCODING`

This attribute defines the data representation used to store graphical objects in an XGL Metafile. The attribute can only be set before any part of the Metafile is written to the output stream, after which the attribute becomes read-only.

XGL supports three standard CGM encoding formats.

- `XGL_CGM_CLEAR_TEXT` uses the 94 printing characters from the ISO 646 7-bit code table to represent data. Designed to be "human-readable" and self-delimiting, clear text encoding is comparatively slow and complex to parse and generate. This is the default value.
- `XGL_CGM_CHARACTER` uses ASCII bytes, including control characters, to represent element codes and parameters. It is self-delimiting, although more compact than clear text encoding. Designed for transmission over communication lines, it is comparatively slow to parse and generate.
- `XGL_CGM_BINARY` uses internally formatted binary numbers to represent data. It is the fastest to read and write, and provides the most compact representation.

`XGL_CGM_DESCRIPTION`

This attribute defines the Metafile header information and the file pointer to the Metafile output file. It can only be set before any part of the Metafile has been written to the output stream, after which the attribute becomes read-only. The Metafile description information is specified by the data type *Xgl_cgm_description*, which has three components:

- *cgm_fp* is the stream pointer where the Metafile should be written. If the file already exists, XGL appends the Metafile to it. If XGL is given a structure with the file pointer equal to NULL, the default value is `stdout`.

- *cgm_signature* is a string containing the Metafile label, which could include a description of the Metafile's contents. If XGL is given a structure with the string pointer equal to NULL, the default value is *XGL 3.0 CGM 1.0 current date and time.*

cgm_description is a string containing descriptive information about the Metafile, such as the author, the place of origin, and date and time of generation. If XGL is given a structure with the string pointer equal to NULL, the default value is: *Sun Microsystems CGM TOP/FULL conformance.*

`XGL_CGM_SCALE_MODE`

This attribute defines the data representation used to store graphical objects in an XGL Metafile. It can only be set before any part of the Metafile has been written to the output stream, after which time the attribute becomes read-only. Two encoding formats are currently supported in XGL as defined by the data type *Xgl_cgm_scale_mode*:

- `XGL_CGM_ABSTRACT` means there is no correspondence between the range of coordinate values in the Metafile and the image it produces. In this case, the SCALEMODE will always be ABSTRACT, and the scale factor will always be 1.0. `XGL_CGM_ABSTRACT` is the default value.
- `XGL_CGM_METRIC` means there is a correspondence between the range of coordinate values in the Metafile and the image it produces. In this case, the SCALEMODE will be METRIC, and the scale factor (set with the attribute `XGL_CGM_SCALE_FACT`), will indicate the mapping between one unit in Metafile coordinate values and one millimeter in the displayed image. This mode allows images of predetermined size to be exchanged using CGM.

`XGL_CGM_SCALE_FACTOR`

This attribute defines the scale factor used when the SCALEMODE delimiter is set to METRIC. It can only be set before any part of the Metafile has been written to the output stream, after which the attribute becomes read-only. The value is the address of a floating point number. The default scale value is 1.0.

`XGL_CGM_PICTURE_DESCRIPTION`

This attribute defines the descriptive label for a picture within an XGL Metafile. It can only be set before any part of the Metafile has been written, after which the attribute becomes read-only. The picture name attribute is described as a string, *Xgl_sgn8 **. This string value is output to the Metafile with the `BEGIN PICTURE` delimiter in the Metafile. This delimiter is written

(and a new picture started) when the user calls xgl_context_new_frame(). Any changes to the picture name attribute after this call does not affect the current picture.

XGL_DEV_COLOR_MAP
This attribute permits the application to set its own Color Map into the Metafile, which will be used to specify the colors rendered into the Metafile. The attribute specifies the handle of the Color Map object to be associated with the Metafile Device. Once the Color Map is created, it is attached to the Metafile using the xgl_object_set() operator and this attribute. Color Map objects can only be attached to Rasters and Metafiles, not Contexts. The default value is a handle to a Color Map object containing a two-element, monochrome (black and white) color table if XGL_DEV_COLOR_TYPE is set to XGL_COLOR_INDEX and a 6-9-4 color cube is XGL_DEV_COLOR_TYPE is set to XGL_COLOR_RGB.

XGL_CGM_VDC_EXTENT
This attribute controls the type and range of the coordinate values. XGL currently supports only integer extent. The values are:

- XGL_CGM_VDC_EXT_SHORT, a 16-bit VDC extent.
- XGL_CGM_VDC_EXT_LONG, a 32-bit VDC extent. This extent is implemented but not officially supported. This is due to overflow problems with transformations of 32-bit integers and floating point numbers in the XGL pipeline.

XGL CGM Line Patterns

CGM Metafiles only support a fixed set of line patterns. The default pattern is solid fill. User-defined line patterns or line pattern types not on the list below will default to solid lines. The line patterns supported are as follows:

```
xgl_lpat_cgm_dotted
xgl_lpat_cgm_dashed
xgl_lpat_dash_dot
xgl_lpat_dash_dot_dotted
xgl_lpat_long_dashed
```

An application must use the XGL attribute XGL_CTX_LINE_PATTERN or XGL_CTX_EDGE_PATTERN to achieve line or edge patterns in the CGM Metafile.

XGL CGM Markers

CGM Metafiles only support a fixed set of marker types. The default marker type is a dot marker. If there is no application data (XGL_OBJ_APPLICATION_DATA) attached to the marker object currently in use, user-defined markers and marker types not on the list below default to dot markers. The marker types supported are as follows:

```
xgl_marker_dot
xgl_marker_plus
xgl_marker_asterisk
xgl_marker_circle
xgl_marker_cross
```

An application must use the XGL attribute XGL_CTX_MARKER to set the current marker type in the CGM Metafile.

An application can place user-defined markers in the CGM Metafile by setting the XGL_OBJ_APPLICATION_DATA attribute of the marker object to point to the marker data, as in the example below:

```
Xgl_marker              my_marker;
Xgl_sys_state           sys_state;
Xgl_pt_list_list        pll;
float                   scale_factor;

sys_state - xgl_open(<attribute-list>);
<Initialize pll for user-defined marker>;
<Set scale factor >;
my_marker = xgl_object_create(sys_state,
                        XGL_MARKER,NULL,
                        XGL_MARKER_DESCRIPTION, &pll,
                        XGL_OBJ_APPLICATION_DATA, &scale_factor,
                        0);
```

When application data is added to a Marker object, XGL will stroke out the marker into lines and scale these lines by a factor equal to the scale factor pointed to by the application data multiplied by XGL_CTX_MARKER_SCALE_FACTOR. Whereas normal markers have a fixed size, stroked out markers will scale as the size of the image producted from the CGM Metafile is changed.

Important Notes on Integrating XGL with a Windowing System

XGL is an output-only library. The application uses the window system's input and event mechanisms to provide interaction with the user. X toolkit and Xlib function calls interact with the OpenWindows environment to create and manipulate windows on the display device, and XGL draws into the X11 windows created by the application. If the canvas has been created using an X toolkit, the application must explicitly get the X window ID from the canvas to associate the canvas with a Window Raster. The Xlib `XCreateWindow` call returns the handle that XGL requires.

Mixing XGL and Xlib Drawing Routines

The DGA facility allows XGL to communicate directly with the graphics hardware, without going through the Xlib communications path to the server. Because of this, application programmers must be careful when mixing XGL and Xlib calls, as latencies involved with XGL communication with graphics hardware and Xlib communication with the server can result in geometry being drawn in an unexpected order. For example, a line drawn by an Xlib function may appear on the screen after a polygon drawn by XGL even though the Xlib function was called first. If the order in which the primitives are drawn is important to an application, the following precautions should be taken when mixing XGL and Xlib functions:

- An application that has finished drawing with XGL primitives should call `xgl_context_flush(ctx,XGL_FLUSH_SYNCHRONIZE)` to ensure that XGL primitives are displayed before calling Xlib functions. This operator should be called even if the `XGL_CTX_DEFERRAL_MODE` is `XGL_DEFER_ASAP`.
- Xlib calls should be flushed with `XSync` before further XGL commands are issued. However, the use of `XSync` may cause some performance degradation.

Note also that when rendering and clearing the window with Xlib calls, the application must clear the window with `xgl_context_new_frame()` before rendering with XGL primitives, even if the window has previously been cleared with Xlib calls.

Window Resize Events

In OpenWindows, window resize events are *not* trapped internally by XGL. The application must explicitly handle these events with the XGL operator `xgl_window_raster_resize()`. This operator causes XGL to resize its Device Coordinate boundaries and, for 3D applications, update the size of the software Z-buffer and accumulation buffer, if applicable.

If the application will run as a DGA application, it should call `XSync()` before calling `xgl_window_raster_resize()` in order to synchronize XGL and the server. XView, and possibly other toolkits, may call the application resize procedure before sending the resize request to the server. This causes DGA to return a window size that is behind by one resize event. Calling `XSync()` before calling `xgl_window_raster_resize()` will minimize this problem.

Note – Be sure to check that the XGL Window Raster has been created before calling the `xgl_window_raster_resize()` operator. An error may occur if a the resize operator is called before the Window Raster is created.

Device Example Programs

The following programs illustrate creating Device objects using Xlib, the XView toolkit, and the OLIT toolkit. In an XGL program, the Xlib or X toolkit calls are interleaved with the XGL calls, and the results of the X and XGL calls share the same drawable area on the screen. The output of all three example programs is the same and is shown in Figure 4-1 on page 75. The source code for the examples follows.

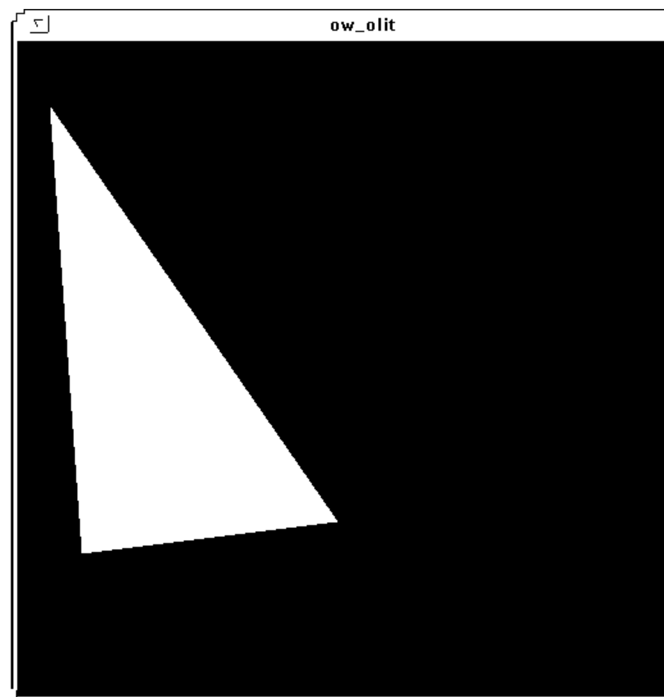


Figure 4-1 Output of `ow_olit.c`

XGL and Xlib

The example program `ow_xlib.c` uses Xlib calls to create the window used by XGL. To compile this program, type `make ow_xlib` in the example program directory.

Code Example 4-1 Using XGL with Xlib

```
/* ow_xlib.c - attach XGL Window Raster to OpenWindows Xlib window */
/*
 * This program demonstrates how to create an XGL Window Raster given
 * an Xlib window.
 */

#include <stdio.h>
#include <X11/Xlib.h>
```

```
#include <X11/Xutil.h>
#include <xgl/xgl.h>

/* need this global so that quit_proc can get it */
static Xgl_objectsys_st;/* XGL System State object */

/* need this global so that repaint procedure can get it */
static Xgl_objectctx = NULL;/* XGL graphics Context object */

main (
    intargc,
    char*argv[])
{
    Xgl_object    win_ras = NULL;/* XGL Window Raster object */
    Xgl_X_window  xgl_x_win;     /* XGL-X data structure */
    Xgl_obj_desc  win_desc;      /* XGL win ras structure */

    Xgl_pt_list   pl;            /* polygon point structure */
    Xgl_pt_i2d    pts_i2d[3];    /* integer 2d points */

    Display       *display;      /* X display data structure */
    int           screen;        /* X screen */
    Visual        *vis;          /* X color visual */
    Window        win;           /* X window */
    XSetWindowAttributes attrs;  /* X create window attributes */
    XEvent        event;         /* X event */

    intdone = 0;/* quit flag */

    /* open X display */
    if ((display = XOpenDisplay (NULL)) == NULL) {
       (void) fprintf (stderr, "cannot open display\n");
       exit (1);
    }

    /* get screen number for this display */
    screen = DefaultScreen (display);

    /* express interest in these events */
    attrs.event_mask = ButtonPressMask | ButtonReleaseMask |
                       ExposureMask;

    /* create window of type visual */
    if (!(win = XCreateWindow (display,
              RootWindow (display, screen),
              10, 10, 500, 500, 1,
```

```
          XDisplayPlanes (display, 0),
          CopyFromParent, CopyFromParent,
          CWEventMask, &attrs))) {
   printf ("can't create visual window\n");
   exit (1);
}
XSetStandardProperties (display, win,
                      "Press any mouse button to quit",
                      "X-Xgl", None, argv, argc, NULL);

/* map and wait for exposure */
XMapWindow (display, win);
do {
   XNextEvent (display, &event);
} while (event.type != Expose);

/* create XGL System State object */
sys_st = xgl_open (NULL);

/* copy X information into XGL data structure */
xgl_x_win.X_display = (void *) display;
xgl_x_win.X_window = (Xgl_usgn32) win;
xgl_x_win.X_screen = (int) screen;

/* create Window Raster Device using XView canvas */
win_desc.win_ras.type = XGL_WIN_X | XGL_WIN_X_PROTO_DEFAULT;
win_desc.win_ras.desc = &xgl_x_win;

win_ras = xgl_object_create (sys_st, XGL_WIN_RAS, &win_desc,
                            XGL_DEV_COLOR_TYPE, XGL_COLOR_INDEX,
                            NULL);

/* create XGL graphics Context object using */
/* the Window Raster object */
ctx = xgl_object_create (sys_st, XGL_2D_CTX, NULL,
        XGL_CTX_DEVICE, win_ras,
        XGL_CTX_DEFERRAL_MODE, XGL_DEFER_ASAP,
        NULL);

/* clear canvas area to default background color of black */
xgl_context_new_frame (ctx);
/* setup data points for a triangle */
/* XGL will close the triangle */
pts_i2d[0].x = 25;
pts_i2d[0].y = 50;
pts_i2d[1].x = 50;
```

```
    pts_i2d[1].y = 400;
    pts_i2d[2].x = 250;
    pts_i2d[2].y = 375;

    /* fill XGL polygon data structure */
    pl.pt_type = XGL_PT_I2D;
    pl.num_pts = 3;
    pl.bbox = 0;
    pl.pts.i2d = &pts_i2d[0];

    /* draw a polygon in the default color of white */
    xgl_polygon (ctx, XGL_FACET_NONE, 0, 0, 1, &pl);

    /* wait for user input and quit on key press event */
    done = 0;
    while (!done) {
       XNextEvent (display, &event);
       if (event.type == Expose) {
           if (!event.xexpose.count) {
               xgl_window_raster_resize(win_ras);
               /* clear rendering area */
               xgl_context_new_frame (ctx);
               /* draw a polygon in default color of white */
               xgl_polygon (ctx, XGL_FACET_NONE, 0, 0, 1, &pl);
           }
       }
       else if (event.type == ButtonPress) {
           done = 1;
       }
    }

    xgl_context_post(ctx, TRUE);

    /* close XGL and free all memory used by XGL */
    xgl_close (sys_st);

    /* destroy X11 resources used */
    XDestroyWindow (display, win);
    XCloseDisplay (display);

    exit (0);
}
```

XGL and the XView Toolkit

The example program `ow_xview.c` uses the XView toolkit to create the window used by XGL. To compile this program, type `make ow_xview` in the example program directory.

Code Example 4-2 Using XGL with XView

```
/* ow_xview.c - attach XGL Window Raster to OpenWindows XView canvas
*/

/*
 * This program demonstrates how to create an XGL Window Raster given
 * an XView canvas.
 */

#include <stdio.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <xview/xview.h>
#include <xview/canvas.h>
#include <xview/xv_xrect.h>

#include <xgl/xgl.h>

/*
 * need this global so that quit_proc can get it
 */
static Xgl_object     sys_st;     /* XGL System State object */

/*
 * need this global so that repaint procedure can get to it
 */
static Xgl_object     ctx = NULL; /* Graphics Context object */
static Xgl_object     win_ras = NULL; /* Xgl window raster */

void    event_proc (Xv_Window, Event*, Notify_arg);
void    repaint_proc (Canvas, Xv_Window, Display*, Window,
                     Xv_xrectlist*);
Notify_value    quit_proc (Frame, Destroy_status);

main (
    int                argc,
    char               *argv[])
```

```
{
    Xgl_X_window     xgl_x_win;  /* XGL-X data structure */
    Xgl_obj_desc     win_desc;   /* Window raster structure */

    Frame            frame;   /* XView frame around window */
    Canvas           canvas;  /* XView canvas in frame */
    Xv_Window        pw;             /* XView paint window */
    Window           frame_window;   /* XID of frame */
    Window           canvas_window;  /* XID of canvas */
    Display          *display;       /* pointer to X display */
    int              screen;         /* X screen number */

    xv_init (XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

  / * create a XView simple window frame - WIN_DYNAMIC_TRUE must be
     * specified for OW 1.0 where the default is static color maps
     */
    frame = xv_create (NULL, FRAME,
                       FRAME_LABEL,      "View Window",
                       WIN_DYNAMIC_VISUAL,     TRUE,
                       OPENWIN_AUTO_CLEAR,     FALSE,
                       CANVAS_RETAINED,        FALSE,
                       CANVAS_FIXED_IMAGE,     FALSE,
                       WIN_DEPTH,              8,
                       XV_VISUAL_CLASS,        PseudoColor,
                       NULL);

    /*
     * create XView canvas - WIN_DYNAMIC_TRUE must be specified OW
     * where the default is static color maps
     */
    /* set event and repaint procedures */
    canvas = xv_create (frame, CANVAS,
                        WIN_DYNAMIC_VISUAL,     TRUE,
                        OPENWIN_AUTO_CLEAR,     FALSE,
                        CANVAS_RETAINED,        FALSE,
                        WIN_EVENT_PROC,         event_proc,
                        CANVAS_REPAINT_PROC,    repaint_proc,
                        WIN_DEPTH,              8,
                        XV_VISUAL_CLASS,        PseudoColor,
                        NULL);

    /* get X stuff */
    display = (Display *) xv_get (frame, XV_DISPLAY);

    pw = (Xv_Window) canvas_paint_window (canvas);
```

```
    canvas_window = (Window) xv_get (pw, XV_XID);
    frame_window  = (Window) xv_get (frame, XV_XID);

    /* put X stuff into XGL data structure */
    xgl_x_win.X_display = (void *) XV_DISPLAY_FROM_WINDOW (pw);
    xgl_x_win.X_window  = (Xgl_usgn32) canvas_window;
    xgl_x_win.X_screen  = (int) DefaultScreen (display);

    /* wait for the window */
    sleep (2);

    /* create XGL System State object */
    sys_st = xgl_open (NULL);

    /* create Window Raster Device using XView canvas */
    win_desc.win_ras.type = XGL_WIN_X;
    win_desc.win_ras.desc = &xgl_x_win;
    win_ras = xgl_object_create (sys_st, XGL_WIN_RAS, &win_desc,
                                NULL);

    /* create XGL Context object using the Window Raster object */
    ctx = xgl_object_create (sys_st, XGL_2D_CTX, NULL,
                             XGL_CTX_DEVICE, win_ras,
                             XGL_CTX_DEFERRAL_MODE, XGL_DEFER_ASAP,
                             NULL);

    /* set quit procedure where xgl_close is called */
    notify_interpose_destroy_func (frame, quit_proc);

    /* go into XView loop */

    exit (0);
}

static
void
event_proc (
    Xv_Window            window,
    Event               *event,
    Notify_arg           arg)
{
    switch (event_action (event)) {
        /*
         * not anything to do here, but mouse and keyboard
         * events should be tracked in this procedure
         */
```

```
    default:
        break;
    }
}

/*
 * Xview repaint procedure.  Draw the polygon in this function.
 */
static
void
repaint_proc (
    Canvas              local_canvas,
    Xv_Window           local_pw,
    Display             *dpy,
    Window              xwin,
    Xv_xrectlist        *xrects)
{
    Xgl_pt_list         pl;  /* polygon point structure */
    Xgl_pt_i2d          pts_i2d[3]; /* integer 2d points */

    xgl_window_raster_resize(win_ras);

    /* clear canvas area to default background color of black */
    xgl_context_new_frame (ctx);

    /* setup data points for a triangle - XGL will close triangle */
    pts_i2d[0].x = 25;
    pts_i2d[0].y = 50;
    pts_i2d[1].x = 50;
    pts_i2d[1].y = 400;
    pts_i2d[2].x = 250;
    pts_i2d[2].y = 375;

    /* fill XGL polygon data structure */
    pl.pt_type = XGL_PT_I2D;
    pl.num_pts = 3;
    pl.bbox = 0;
    pl.pts.i2d = &pts_i2d[0];

    /* draw a polygon in the default color of white */
    xgl_polygon (ctx, XGL_FACET_NONE, 0, 0, 1, &pl);
}
static
Notify_value
quit_proc (
    Frame               fr,
```

```
    Destroy_status      status)
{
    if (status == DESTROY_CHECKING) {
        xgl_close (sys_st);
    }
    return (notify_next_destroy_func (fr, status));
}
```

XGL and the OLIT Toolkit

The example program `ow_olit.c` uses the OLIT toolkit calls to create the window used by XGL. To compile this program, type `make ow_olit` in the example program directory.

Code Example 4-3 Using XGL with OLIT

```
/* ow_olit.c - attach XGL Window Raster to OLIT stub widget */

/*
 * This program demonstrates how to create an XGL Window
 * Raster given an OLIT stub widget.
 */

#include <stdio.h>
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <Xol/OpenLook.h>
#include <Xol/DrawArea.h>
#include <xgl/xgl.h>

/*
 * need these global so that expose procedure can get to them
 */
static Xgl_object     ctx = NULL;/* XGL Context object */
static Xgl_object     sys_st;    /* XGL System State object */
static Xgl_object     win_ras = NULL; /* XGL Window Raster */

main (
    int               argc;
    char              *argv[])
{
    Xgl_X_window      xgl_x_win; /* XGL-X data structure */
    Xgl_obj_desc      win_desc;  /* Window raster struct */
```

```
Widget            toplevel;   /* Widget top-level */
Widget            drawable;   /* Drawable widget */
XtAppContext      app;
Window            window;     /* X window for canvas */
Display           *display;   /* pointer to X display */

void              repaint(); /* OLIT repaint procedure */
/*
 * Initialize toolkits
 */
OlToolkitInitialize((XtPointer)NULL);
toplevel = XtVaAppInitialize (&app, "ow_olit",
          (XrmOptionDescList)NULL, 0,
          &argc, argv, (String *)NULL,
          XtNtitle, "Simple OLIT Drawable",
          XtVaTypedArg, XtNvisual, XtRString,
          "PseudoColor", sizeof("PseudoColor"),
          NULL);

/*
 * Create a draw area widget for rendering graphics
 * and add expose callback to draw area widget
 */
drawable = XtVaCreateManagedWidget("drawable",
          drawAreaWidgetClass, toplevel,
          XtNlayout, OL_IGNORE,
          XtNheight, 512,
          XtNwidth,  512,
          NULL);
XtAddCallback(drawable, XtNexposeCallback, repaint,
              (XtPointer) NULL);

XtRealizeWidget(toplevel);

/* get X stuff */
display = (Display *) XtDisplay (drawable);
window = (Window) XtWindow (drawable);

/* put X stuff into XGL data structure */
xgl_x_win.X_display = (void *) display;
xgl_x_win.X_window  = (Xgl_usgn32) window;
xgl_x_win.X_screen  = (int) DefaultScreen (display);

/* wait for the window */
sleep (2);
```

```
    /* create XGL System State object */
    sys_st = xgl_open (NULL);

    /* create Window Raster Device using OLIT widget */
    win_desc.win_ras.type = XGL_WIN_X;
    win_desc.win_ras.desc = &xgl_x_win;
    win_ras = xgl_object_create (sys_st, XGL_WIN_RAS, &win_desc,
                                 NULL);

    /* create XGL Context object using the Window Raster object */
    ctx = xgl_object_create (sys_st, XGL_2D_CTX, NULL,
                             XGL_CTX_DEVICE, win_ras,
                             XGL_CTX_DEFERRAL_MODE, XGL_DEFER_ASAP,
                             NULL);

    /* go into Xt main loop and wait for expose events */
    XtAppMainLoop(app);

    exit (0);
}

/*
 * Xgl repaint procedure called on expose events
 */
static
void
repaint ()
{
    Xgl_pt_list       pl;        /* polygon point structure */
    Xgl_pt_i2d        pts_i2d[3];/* integer 2d points */

    /* clear draw area to default background color of black */
    xgl_context_new_frame (ctx);

    /* set up data points for a triangle - XGL will close triangle */
    pts_i2d[0].x = 25;
    pts_i2d[0].y = 50;
    pts_i2d[1].x = 50;
    pts_i2d[1].y = 400;
    pts_i2d[2].x = 250;
    pts_i2d[2].y = 375;

    /* fill XGL polygon data structure */
    pl.pt_type = XGL_PT_I2D;
    pl.num_pts = 3;
```

```
    pl.bbox = 0;
    pl.pts.i2d = &pts_i2d[0];

    /* draw a polygon in the default color of white */
    xgl_polygon (ctx, XGL_FACET_NONE, 0, 0, 1, &pl);
}
```

Determining Device Acceleration

XGL accelerates a primitive if some portion of the geometry is processed directly by accelerating hardware. Acceleration can speed up geometric transformations and clipping, as well as actual rendering. To determine what acceleration features underlie a window, an application can use the `xgl_inquire()` operator. `xgl_inquire()` is defined as:

```
Xgl_inquire* xgl_inquire(
    Xgl_sys_state       sys_state
    Xgl_obj_desc        *desc);
```

The parameter `desc` contains window information. The operator returns a pointer to a structure of type *Xgl_inquire*. This structure contains the following fields of information about the frame buffer underlying the window:

Table 4-2 Fields Returned by `xgl_inquire()`

Field	Information
Name	The XGL name of the frame buffer underlying the window.
DGA flag	Specifies whether XGL does direct graphics access to render pixels into the frame buffer underlying the window. If DGA is not available, XGL uses Xlib or PEXlib to render pixels.
Color type	Indicates whether the frame buffer is color index or RGB.
Depth	Specifies the depth of the frame buffer underlying the window.
Width	Specifies the width of the frame buffer in pixel coordinates.
Height	Specifies the height of the frame buffer in pixel coordinates.
Maximum buffer	Specifies the maximum number of buffers available from the frame buffer underlying the window. A value of two means the application might be able to perform double buffering.

Table 4-2 Fields Returned by `xgl_inquire()` *(Continued)*

Field	Information
Double buffer is copy	Specifies whether the underlying hardware device performs double buffering by copying pixels from the draw (hidden) buffer to the display buffer.
Point type	Specifies the point types supported by the frame buffer.
HLHSR mode	Indicates whether the frame buffer underlying the window directly supports a Z-buffer and Z-buffer rendering.
Picking Double buffer Indexed color True color Depth cueing Lighting Shading Antialiasing HLHSR	These fields reflect the capabilities of the underlying hardware accelerator, if any. If the accelerator does not support this functionality, these fields are set to XGL_INQ_NOT_SUPPORTED. If support for the functionality is provided through software emulation, these fields are set to XGL_INQ_SOFTWARE. If hardware acceleration is provided, these fields are set to XGL_INQ_HARDWARE.
Stereo	Specifies whether the device underlying the window raster has hardware support for stereo.
Extensions	Bitmask indicating the extensions supported by the frame buffer underlying the window. Currently, only queries for the multibuffer extension to X (MBX). To determine whether MBX is supported, AND the value of this field with the bitmask XGL_INQ_MULTI_BUFFERING. If the result is not zero, the frame buffer supports MBX.
Transparency	Specifies whether transparency is supported in hardware or in software.

Note – The application must open XGL before calling `xgl_inquire()`. Device characteristics cannot be queried before XGL is open.

Note – The `xgl_inquire()` operator does not currently return information on accumulation.

Inquire Example Program

The program `inq.c` shows the use of `xgl_inquire()`. To compile this program, type `make inq` in the example program directory. An example of program output for a particular hardware device is shown in Figure 4-2. The source code for the example follows.

```
% inq
Inquire Information:
  Sun:GX
  Direct Graphics Access
    Color Model: INDEXED
    Depth : 8
    Width : 1152
    Height: 900
    Single Buffering
      Using bitblt transfer
    Point Types:
        2D
        3D
        INT
        FLOAT
    Software Zbuffer in host
    Picking emulated in software
    Double buffering emulated in software
    Index colors accelerated through hardware
    True colors emulated in software
    Depth cueing emulated in software
    Lighting emulated in software
    Shading emulated in software
    Hidden line removal emulated in software
    Antialiasing emulated in software
    No stereo support
    No Multi Buffering support
    Transparency emulated in software
%
```

Figure 4-2 Output of `inq.c`

Code Example 4-4 Inquire Example

```
/*
 * inq.c
 */

#include <stdio.h>
#include <math.h>
#include <xgl/xgl.h>
#include <X11/Xlib.h>

/* Xgl inquiry function */

/*
 * main routine
 *      description:
 *          This example program opens an Xlib display and window
 *          and uses the Xgl inquiry function to determine the
 *          hardware features underlying that window.
 *      arguments:
 *              -display [X display number]
 */

static  char    *colon_zero = ":0";

main(
    int                 argc,
    char                *argv[])
{
    Xgl_object          sys_st;
    Xgl_obj_desc        obj_desc; /* XGL window system information */
    Xgl_inquire         *inq_info;     /* Xgl inquiry structure */
    Xgl_X_window        xgl_x_win;     /* Xgl to X structure */
    char                *display_name; /* name of X display */
    Display             *display;      /* Xlib display structure */
    int                 screen;        /* screen for X display */
    Window              window;        /* X window we create */

    if (argc < 2)
        display_name = colon_zero;
    else
        display_name = argv[2];

    display = XOpenDisplay(display_name);
```

```
    if (display == (Display *)NULL) {
        printf("could not open display %s\n", display_name);
        exit(1);
    }
    screen = DefaultScreen(display);

 window = XCreateSimpleWindow(
        display,                      /* Xlib display structure */
        RootWindow(display, screen),  /* use X root window */
        0, 0,                         /* x, y position */
        1, 1,                         /* width, height */
        1,                            /* border width in pixels */
        BlackPixel(display, screen),  /* border color */
        BlackPixel(display, screen)); /* background color */

    if (window == 0) {
        printf("could not open window %s\n", display_name);
        exit(1);
    }

    sys_st = xgl_open (XGL_SYS_ST_ERROR_DETECTION, TRUE,
                       NULL);

    xgl_x_win.X_display = (void *)display;
    xgl_x_win.X_screen = screen;
    xgl_x_win.X_window = window;

    obj_desc.win_ras.type = XGL_WIN_X;
    obj_desc.win_ras.desc = &xgl_x_win;
    if (!(inq_info = xgl_inquire(sys_st, &obj_desc))) {
      printf("error getting inquiry\n");
      exit(1);
    }

    printf("Inquire Information:\n");

    printf("  %s\n", inq_info->name);
    if (inq_info->dga_flag)
        printf("  Direct Graphics Access\n");
    else
        printf("  Xlib port\n");

    printf("    Accelerated Color Type(s): ");
    if (!inq_info->color_type.index &&
        !inq_info->color_type.rgb)
```

```
    printf("None\n");
else {
    if (inq_info->color_type.index)
        printf("INDEXED ");
    if (inq_info->color_type.rgb)
      printf("RGB ");
    printf("\n");
}
printf("    Depth : %d\n", inq_info->depth);
printf("    Width : %d\n", inq_info->width);
printf("    Height: %d\n", inq_info->height);
if (inq_info->maximum_buffer == 1)
    printf("    Single Buffering\n");
if (inq_info->maximum_buffer == 2)
    printf("    Double Buffering\n");
if (inq_info->maximum_buffer > 2)
    printf("    Multiple Buffering (%d)\n",
                 inq_info->maximum_buffer);
if (inq_info->db_buffer_is_copy)
    printf("      Using bitblt transfer\n");
else
    printf("      Using hardware swap\n");
printf("    Point Types:\n");
if (!inq_info->pt_type.pt_dim_2d &&
    !inq_info->pt_type.pt_dim_3d &&
    !inq_info->pt_type.pt_type_int &&
    !inq_info->pt_type.pt_type_float) {
  printf("        None (?)\n");
}
else {
  if (inq_info->pt_type.pt_dim_2d)
      printf("        2D\n");
  if (inq_info->pt_type.pt_dim_3d)
      printf("        3D\n");
  if (inq_info->pt_type.pt_type_int)
      printf("        INT\n");
  if (inq_info->pt_type.pt_type_float)
      printf("        FLOAT\n");
}
if (inq_info->hlhsr_mode == XGL_HLHSR_NONE)
    printf("    Software Zbuffer in host\n");
if (inq_info->hlhsr_mode == XGL_HLHSR_ZBUFFER)
    printf("    Hardware Zbuffer\n");

if (inq_info->picking == XGL_INQ_NOT_SUPPORTED)
    printf("    Picking not supported\n");
```

```
    else if (inq_info->picking == XGL_INQ_SOFTWARE)
        printf("    Picking emulated in software\n");
    else if (inq_info->picking == XGL_INQ_HARDWARE)
        printf("    Picking accelerated through hardware\n");

    if (inq_info->double_buffer == XGL_INQ_NOT_SUPPORTED)
        printf("    Double buffering not supported\n");
    else if (inq_info->double_buffer == XGL_INQ_SOFTWARE)
        printf("    Double buffering emulated in software\n");
    else if (inq_info->double_buffer == XGL_INQ_HARDWARE)
      printf("   Double buffering accelerated through hardware\n");

    if (inq_info->indexed_color == XGL_INQ_NOT_SUPPORTED)
        printf("    Index colors not supported\n");
    else if (inq_info->indexed_color == XGL_INQ_SOFTWARE)
        printf("    Index colors emulated in software\n");
    else if (inq_info->indexed_color == XGL_INQ_HARDWARE)
        printf("    Index colors accelerated through hardware\n");

    if (inq_info->true_color == XGL_INQ_NOT_SUPPORTED)
        printf("    True colors not supported\n");
    else if (inq_info->true_color == XGL_INQ_SOFTWARE)
        printf("    True colors emulated in software\n");
    else if (inq_info->true_color == XGL_INQ_HARDWARE)
        printf("    True colors accelerated through hardware\n");

    if (inq_info->depth_cueing == XGL_INQ_NOT_SUPPORTED)
        printf("    Depth cueing not supported\n");
    else if (inq_info->depth_cueing == XGL_INQ_SOFTWARE)
        printf("    Depth cueing emulated in software\n");
    else if (inq_info->depth_cueing == XGL_INQ_HARDWARE)
        printf("    Depth cueing accelerated through hardware\n");

    if (inq_info->lighting == XGL_INQ_NOT_SUPPORTED)
        printf("    Lighting not supported\n");
    else if (inq_info->lighting == XGL_INQ_SOFTWARE)
        printf("    Lighting emulated in software\n");
    else if (inq_info->lighting == XGL_INQ_HARDWARE)
        printf("    Lighting accelerated through hardware\n");

  if (inq_info->shading == XGL_INQ_NOT_SUPPORTED)
        printf("    Shading not supported\n");
    else if (inq_info->shading == XGL_INQ_SOFTWARE)
        printf("    Shading emulated in software\n");
    else if (inq_info->shading == XGL_INQ_HARDWARE)
        printf("    Shading accelerated through hardware\n");
```

```
    if (inq_info->hlhsr == XGL_INQ_NOT_SUPPORTED)
        printf("    Hidden line removal not supported\n");
    else if (inq_info->hlhsr == XGL_INQ_SOFTWARE)
        printf("    Hidden line removal emulated in software\n");
    else if (inq_info->hlhsr == XGL_INQ_HARDWARE)
      printf(" Hidden line removal accelerated through hardware\n");

    if (inq_info->antialiasing == XGL_INQ_NOT_SUPPORTED)
        printf("    Antialiasing not supported\n");
    else if (inq_info->antialiasing == XGL_INQ_SOFTWARE)
        printf("    Antialiasing emulated in software\n");
    else if (inq_info->antialiasing == XGL_INQ_HARDWARE)
        printf("    Antialiasing accelerated through hardware\n");

    if (inq_info->stereo)
        printf("    Hardware stereo support\n");
    else
        printf("    No stereo support\n");

   if (inq_info->extns & XGL_INQ_MULTI_BUFFERING)
       printf("   Multi Buffering supported\n");
   else
       printf("   No Multi Buffering support\n");

   if (inq_info->transparency == XGL_INQ_NOT_SUPPORTED)
        printf("    Transparency not supported\n");
    else if (inq_info->transparency == XGL_INQ_SOFTWARE)
        printf("    Transparency emulated in software\n");
    else if (inq_info->transparency == XGL_INQ_HARDWARE)
        printf("    Transparency accelerated through hardware\n");

    free (inq_info);

}
```

Transparent Overlay Windows

The XGL library supports the use of transparent overlay windows. These windows are X windows that allow the user to see through to the underlying window on a per-pixel basis. Transparent overlay windows allow applications to render temporary imagery on complex rendering in the underlying window. Users of an application that provides transparent overlay windows can annotate an image with text or graphical figures, temporarily highlight certain portions of the imagery, or animate figures that appear to move against the background of the imagery. When geometry in the overlay is cleared, any underlying graphics do not need to be regenerated.

Applications can draw into transparent overlay windows with *transparent* paint or *opaque* paint. Pixels rendered transparently have no intrinsic color; they derive their displayed color from the pixels that lie beneath. Pixels rendered opaquely obscure pixels in underlying windows. This allows the application to selectively erase the contents of the overlay window in such a way that the graphics in the underlay window show through. Overlay window background is transparent by default but may be set to other types of background.

An XGL application can render into overlay windows using XGL primitives or Xlib primitives.The XGL library provides Context attributes that enable an XGL application to render XGL geometry into an overlay window using opaque or transparent paint.

The transparent overlay support provided in the XGL library requires the transparent overlay API in the Solaris Overlay extension to the Solaris X server. For details on how to verify that the server includes the overlay extension, see the *Solaris X Window System Developer's Guide*, which is part of the Software Developer Kit AnswerBook.

Performance Cases for Transparent Overlays

If the server has overlay support, application geometry can be rendered to the overlay window using hardware overlay planes or software emulation of overlay functionality. Performance may vary considerably between hardware and software overlay rendering. Although the application cannot choose the rendering path, the following information may help the application understand performance differences.

- Hardware overlay planes and XGL/DGA support – If the hardware device provides hardware overlay planes and and the application chooses the correct visuals, the Solaris server can make use of the hardware overlay functionality for transparent overlay windows. If the device graphics handler implements overlay support, XGL coordinates with the server to provide accelerated rendering into transparent overlay windows using DGA. The ZX and SX accelerators are examples of devices that have hardware overlay planes and DGA rendering.

  Rendering through XGL/DGA to hardware overlay planes provides the best performance for rendering to overlay windows. In addition, DGA rendering to both the overlay and underlay windows ensures that images will touch the same pixels. However, there are restrictions that application programs must follow to enable this case. These restrictions are listed on page 97.

- Hardware overlay planes and Xlib support – If the hardware device provides hardware overlay planes, but the device graphics handler does not implement DGA overlay support, XGL renders to the overlay window using Xlib. Even if DGA support of overlays is available, there may be some cases in which the server refuses to grant the request for the overlay window. For example, the server will always deny access to an overlay Drawable on a window with backing store, and in these cases XGL will use Xlib for overlay rendering. In addition, if the application chooses not to follow the restrictions listed on page 97, Xlib is used to render to the overlay window.

  Rendering through Xlib to hardware overlay planes provides moderate performance. However, when geometry is rendered to the underlay window with DGA and to the overlay window with Xlib, there may be cases in which different pixels are touched.

- Software emulation – Even if the hardware does not have overlay planes, the Solaris server can create and use transparent overlay windows if the server includes overlay visuals. In this case, the server emulates overlay functionality in software, and XGL renders using Xlib.

  The GX and p9000 devices are examples of devices that provide overlay functionality through software emulation. Software emulation of overlay windows provides slower performance for overlay rendering.

Information on whether XGL is using DGA to render to the overlay window is not available from within the application, since `xgl_inquire()` does not currently return information on device support for transparent overlays. If the overlay rendering is slow, you may be able to infer that software emulation is

being used. For more accurate information, see your frame buffer documentation for the capabilities of the specific hardware devices, and try to make your application's use of overlay windows as portable as possible. Information on portability issues is provided in the following sections.

Creating an Overlay Window

An application creates an overlay window using the Solaris overlay API `XSolarisOvlCreateWindow`. This routine behaves exactly as `XCreateWindow` except that the resulting window is an overlay window. The newly created window can be rendered into with both opaque and transparent paint. An overlay window must be the child of another window, called an underlay window.

As part of creating an overlay window, the application must search for appropriate visuals for the underlay and overlay windows. Each X screen supporting the overlay extension defines a set of overlay visuals whose windows are best suited for use as children of underlay windows. For each underlay visual, there is a set of optimal overlay visuals. Together, all combinations of underlay visuals and their optimal overlay visuals form the set of optimal overlay/underlay pairs for that screen. The overlay and underlay visuals of an optimal pair are partners of each other.

Choosing Overlay Visuals

The Solaris overlay routine `XSolarisOvlSelectPartner` allows the application to select for a particular underlay visual an optimal overlay visual that meets certain criteria. This routine searches through the optimal partners of the given visual, applying the criteria specified. It returns a success or failure status depending on whether it finds a visual that meets the criteria.

The `XSolarisOvlSelectPartner` routine is designed to help applications construct portable overlay applications. Because of the wide variety of hardware devices, constructing a portable overlay application may be difficult unless care is taken with the choice of visuals. For example, for a given type of underlay window, some devices can provide overlay windows with high-performance rendering, while other devices provide the same type of overlay window but with slower rendering. The `XSolarisOvlSelectPartner` routine enables the application to negotiate with the system for a suitable

overlay/underlay visual pair. For more information on the `XSolarisOvlSelectPartner` routine, see the *Solaris X Window System Developer's Guide*.

Tips on Choosing Overlay Visuals for DGA Transparent Overlays

Some SMCC 3D accelerators, for example, the ZX and the SX, provide accelerated transparent overlay functionality using hardware overlay planes and DGA. However, the application must follow several restrictions to use DGA transparent overlays for XGL rendering:

1. The primary restriction is that the overlay window must always spatially occupy the same pixels as the underlay window. It is the responsibility of the XGL application to maintain this relationship.
2. The overlay window must always be a descendent of the original underlay window and must not be reparented to a different underlay window.
3. To get the appropriate overlay visuals for a frame buffer, use the Solaris overlay API `XSolarisOvlSelectPartner` with a selection type of `XSolarisOvlSelectBestOverlay`. See the program on page 100 for an example of the use of this routine.
4. No windows may be inserted between the overlay and underlay windows.

Note that these restrictions only apply when overlay windows are handled by the hardware device. However, as noted above, the application has no way of determining whether the device that the application is currently rendering on has DGA transparent overlay support because the XGL operator `xgl_inquire()` does not currently return this information. Therefore, to ensure portability, the application may want to abide by these hardware-specific restrictions.

Creating XGL Window Rasters for Overlay Windows

When an application has determined that the server provides overlay support and that the appropriate visuals are available, it can create XGL window rasters for the XGL overlay and underlay windows by copying the X information for these windows into XGL window descriptor structures and creating XGL Window Raster objects. The XGL Window Raster objects for overlay windows are created like any other XGL Window Raster.

Note – To improve the portability of your application and to minimize color flashing, use color maps with the same colors in both the overlay and underlay window color maps. If your application uses pixel-sharing overlay/underlay pairs, create a single color map for both windows. For more information on color maps and overlay windows, see the *Solaris X Window System Developer's Guide*.

Rendering to an Overlay Window Using the XGL API

In the transparent overlay model, the pixels, or *paint*, has a characteristic called the paint type. The paint type can be either opaque or transparent. In an XGL application, the paint type is set on the Context object using the `XGL_CTX_PAINT_TYPE` attribute. This attribute can be set to `XGL_PAINT_OPAQUE`, which is the default value, or `XGL_PAINT_TRANSPARENT`. The attribute `XGL_CTX_PAINT_TYPE` is in effect only when the Device associated with a Context is an overlay window; otherwise, `XGL_CTX_PAINT_TYPE` is ignored, and the paint type is opaque.

Most XGL primitives can render pixels into overlay windows. Note, however, the following exceptions:

- `xgl_context_accumulate()` – Only the color information of the draw buffer is accumulated into an overlay window that is the destination buffer of the accumulation. The paint type is not changed.
- `xgl_context_copy_buffer()` – If the Device associated with the Context is an overlay window, the color information of the source raster is copied to the destination raster as follows:
  - If the source raster is an overlay window and the destination raster is also an overlay window, the color and paint type are copied to the destination.
  - If the source raster is an overlay window but the destination raster is not an overlay window, the color is copied to the destination. If the source paint type is transparent, the color is undefined.

  In addition, if the source raster is not an overlay window but the destination raster is an overlay window, the pixel color is copied, and the paint type of the destination is not changed.

Clearing the Overlay Window Using the XGL API

The XGL attribute XGL_CTX_NEW_FRAME_PAINT_TYPE specifies the paint type for new frame clear actions on overlay windows. This attribute is applied when an application calls xgl_context_new_frame() and XGL_CTX_NEW_FRAME_ACTION includes XGL_CTX_NEW_FRAME_CLEAR. For overlay window clears, the color information is set to the background color, and the paint type is set to the value of XGL_CTX_NEW_FRAME_PAINT_TYPE. By default, the new frame paint type is XGL_PAINT_TRANSPARENT, which clears the overlay window.

Note that when the overlay is handled by the hardware device, erasing the overlay graphics does not damage the underlying graphics or generate an Expose event. This provides a performance advantage when the underlying graphics is complex and requires much time to repaint. However, when the overlay is in software, erasing the overlay graphics may generate an Expose event.

Transparent Overlay Example Program

The example program transparency_ovl.c shows the use of XGL primitives to render to overlay and underlay windows. The program draws an XGL multisimple polygon in the underlay window. It provides a pop-up window whose buttons the user can press to draw XGL text, markers, polylines, or filled polygons in the overlay window. The example program works on hardware that has DGA overlay support, for example the ZX and the SX, as well as on the GX, which does not have hardware overlay planes.

The program uses XSolarisOvlSelectPartner to choose the best overlay and underlay visuals for the screen. If the appropriate visuals are available, the program creates the overlay window with XSolarisOvlCreateWindow and sets the background of the overlay window to transparent with the XSolarisOvlSetWindowTransparent routine. Then the program opens XGL and sets up the XGL window raster descriptor structures for the overlay and underlay windows. The program sets up XGL Contexts for the windows and defines the paint type in the overlay Context as opaque using XGL_CTX_PAINT_TYPE set to XGL_PAINT_OPAQUE. The pop-up window buttons illustrate clearing the overlay window to transparency by calling xgl_context_new_frame() for the overlay Context as well as undrawing the overlay geometry by setting the overlay window paint type to XGL_PAINT_TRANSPARENT.

The sample code below is an excerpt from the complete `transparency_ovl.c` program. See the example program in `$XGLHOME/demo/examples` for the code preceding main.

Note – In this program, color flashing will occur in windows with a depth of 8 bits. You can minimize the color map flashing by offsetting the color map entries. See the example program `color_cube.c` on page 135. Color map flashing does not occur in 24-bit windows.

Code Example 4-5 Transparent Overlay Example

```
main (argc, argv)

    int argc;
    char *argv[];
{
    int     i;
    int     depth;
    int     screen;
    int     buffers;
    int     rotate;
    int     data_count;
    float   x_diff;
    float   y_diff;
    float   x_deg;
    float   y_deg;
    char   *version;
    long    plane_masks[2];
    long    event_mask;
    unsigned long failures;
    Arg     args[8];
    Widget  toplevel;
    Widget  control1;
    Widget  control2;
    Widget  button1;
    Widget  button2;
    Widget  button3;
    Widget  button4;
    Widget  button5;
    Widget  button6;
    Widget  button7;
    Widget  button8;
    Widget  button9;
```

```
    Widget  popup_shell;
    Widget  popup_shell2;
    Widget  underlay;
    Window  ol_window;
    Display *display = NULL;
    Visual  *visual;
    Visual  *default_visual;
    Visual  *ol_visual;
    Screen  *screen_ptr;
    XtAppContext app_context;
    XColor  gray;
    XColor  light_gray;
    XEvent  event;
    XVisualInfo info;
    XSolarisOvlVisualCriteria criteria;
    XSolarisOvlSelectStatus status;
    XSetWindowAttributes attr;
    XClientMessageEvent *client_message =
                 XClientMessageEvent*) &event;
    XVisualInfo visual_info;
    Xgl_segment segment;

/*
 *  Initialize the Openlook toolkit.
 */
    OlToolkitInitialize (NULL);

/*
 *  Create the top level shell.
 */
    toplevel = XtVaAppInitialize (&app_context,
                                  "",
                                  NULL,
                                  0,
                                  (Cardinal*)&argc,
                                  argv,
                                  NULL,
                                  NULL);
    ul_display = (Display*) XtDisplay (toplevel);
    display = ul_display;

/*
 *  Get the screen number.
 */
    screen = DefaultScreen (display);
    root = RootWindow (display, screen);
```

```
/*
 *  Get the default colormap.
 */
    default_colormap = DefaultColormap (display, screen);

/*
 *  Create a popup shell to hold the controls.
 */
    XtSetArg (args[0], XtNx, 100);
    XtSetArg (args[1], XtNy, 100);
    popup_shell = XtCreatePopupShell ("  ",
                                      topLevelShellWidgetClass,
                                      toplevel,
                                      args,
                                      2);

    control1 = XtVaCreateManagedWidget ("control1",
                                        controlAreaWidgetClass,
                                        popup_shell,
                                       XtNlayoutType, OL_FIXEDROWS,
                                        XtNmeasure, 2,
                                        NULL);

    control2 = XtVaCreateManagedWidget ("control",
                                        controlAreaWidgetClass,
                                        control1,
                                       XtNlayoutType, OL_FIXEDCOLS,
                                        XtNmeasure, 1,
                                        NULL);

/*
 *  Create the buttons.
 *  ...
 *  See the example program transparency_ovl.c for details.
 */

/*
 *  Attach the callbacks to the buttons.
 *  ...
 *  See the example program transparency_ovl.c for details.
 */

/*
 * Pop the shell.
 */
```

```
    XtRealizeWidget (popup_shell);
    XtPopup (popup_shell, XtGrabNone);

/*
 *  Create the colormap.
 */
    if (XMatchVisualInfo (display, screen, 24, TrueColor,
                          &visual_info)) {

   /*
    *  Set the visual, color flag and depth.
    */
       visual = visual_info.visual;
       truecolor = TRUE;
       depth = 24;

   /*
    *  Create a colormap for the 24-bit visual.
    */
       colormap = XCreateColormap (display,
                                   RootWindow (display, screen),
                                   visual,
                                   AllocNone);
    }
    else {

   /*
    *  For index color, use the default visual.
    */
       visual = DefaultVisual (display, screen);

   /*
    *  Set the color flag, colormap and depth.
    */
       truecolor = FALSE;
       colormap = default_colormap;
       depth = 8;
    }
    XSync (display, FALSE);

/*
 *  Create a popup shell to hold the graphics widget.
 *  Specify the visual, depth and colormap.
 */
    XtSetArg (args[0], XtNvisual, visual);
    XtSetArg (args[1], XtNdepth, depth);
```

```
    XtSetArg (args[2], XtNcolormap, colormap);
    XtSetArg (args[3], XtNx, 250);
    XtSetArg (args[4], XtNy, 100);
    popup_shell2 = XtCreatePopupShell ("XGL with Overlay",
                                       topLevelShellWidgetClass,
                                       toplevel,
                                       args,
                                       5);

/*
 *  Create a "canvas" and attach it to the popup shell.
 */
    underlay = XtVaCreateManagedWidget ("underlay",
                                        drawAreaWidgetClass,
                                        popup_shell2,
                                        XtNwidth,  400,
                                        XtNheight, 400,
                                        NULL);
/*
 *  Pop the shell.
 */
    XtRealizeWidget (popup_shell2);
    XtPopup (popup_shell2, XtGrabNone);
    ul_window = (Xgl_usgn32) XtWindow (underlay);

/*
 *  Create the overlay window.
 *  Choose the best overlay visual for the underlay window.
 */
    criteria.hardCriteriaMask = 0;
    criteria.softCriteriaMask = XSolarisOvlPreferredPartner;
    status = XSolarisOvlSelectPartner (display,
                                       screen,
                                      XVisualIDFromVisual(visual),
                                    XSolarisOvlSelectBestOverlay,
                                       2,
                                       &criteria,
                                       &info,
                                       &failures);

    if (status != XSolarisOvlSuccess) {
      printf ("Cannot find preferred partner overlay visual...\n");
       if (status != XSolarisOvlQualifiedSuccess) {
           printf("No suitable visuals found:  Exiting.\n");
           exit (0);
       } else
```

```
            printf("Using closest matching visual for overlay.\n");
    }

/*
 * Create the colormap for the overlay window.
 */
    ol_visual = info.visual;

    ol_colormap = XCreateColormap(display, RootWindow(display,
                  screen), ol_visual, AllocNone);

    XAllocColorCells (display, ol_colormap, (Bool) TRUE,
                      plane_masks, 0,
                      ol_color_mapping, 5);

    xcolor[0].flags = DoRed | DoGreen | DoBlue;
    xcolor[0].pixel = ol_color_mapping[0];
    xcolor[0].red   = 65535;
    xcolor[0].green = 65535;
    xcolor[0].blue  = 65535;

    xcolor[1].flags = DoRed | DoGreen | DoBlue;
    xcolor[1].pixel = ol_color_mapping[1];
    xcolor[1].red   =     0;
    xcolor[1].green =     0;
    xcolor[1].blue  =     0;

    xcolor[2].flags = DoRed | DoGreen | DoBlue;
    xcolor[2].pixel = ol_color_mapping[2];
    xcolor[2].red   = 65535;
    xcolor[2].green =     0;
    xcolor[2].blue  =     0;

    xcolor[3].flags = DoRed | DoGreen | DoBlue;
    xcolor[3].pixel = ol_color_mapping[3];
    xcolor[3].red   =     0;
    xcolor[3].green = 65535;
    xcolor[3].blue  =     0;

    xcolor[4].flags = DoRed | DoGreen | DoBlue;
    xcolor[4].pixel = ol_color_mapping[4];
    xcolor[4].red   =     0;
    xcolor[4].green =     0;
    xcolor[4].blue  = 65535;

    XStoreColors (display, ol_colormap, xcolor, 5);
```

```
    XSync (display, FALSE);

    attr.border_pixel = xcolor[0].pixel;
    attr.colormap = ol_colormap;
    attr.event_mask = ExposureMask;

/*  Create the overlay window -
 *  1. The parent underlay window is ul_window.
 *  2. The postion is 0,0 relative to the underlay window.
 *  3. The overlay window is the same size (400,400) as the
 *     underlay window.
 */
    ol_window = XSolarisOvlCreateWindow(display,
                              ul_window,
                              0, 0,
                              400, 400,
                              0,
                              8,
                              InputOutput,
                              ol_visual,
                              CWEventMask |
                            CWBorderPixel |
                              CWColormap,
                              &attr);

/* Set the overlay window background to be transparent. */

    XSolarisOvlSetWindowTransparent (display, ol_window);

/*
 * Define an 8-bit colormap if the underlay window is a
 * Pseudocolor visual.
 */
    if (!truecolor) {
        XAllocColorCells (display, default_colormap,
                         (Bool) TRUE, plane_masks, 0,
                         color_mapping, 128);

        xcolor[0].flags = DoRed | DoGreen | DoBlue;
        xcolor[0].pixel = color_mapping[0];
        xcolor[0].red   = 65535;
        xcolor[0].green = 65535;
        xcolor[0].blue  = 65535;

        xcolor[1].flags = DoRed | DoGreen | DoBlue;
```

```
        xcolor[1].pixel = color_mapping[1];
        xcolor[1].red   =     0;
        xcolor[1].green =     0;
        xcolor[1].blue  =     0;

        xcolor[2].flags = DoRed | DoGreen | DoBlue;
        xcolor[2].pixel = color_mapping[2];
        xcolor[2].red   = 65535;
        xcolor[2].green =     0;
        xcolor[2].blue  =     0;

        xcolor[3].flags = DoRed | DoGreen | DoBlue;
        xcolor[3].pixel = color_mapping[3];
        xcolor[3].red   =     0;
        xcolor[3].green = 65535;
        xcolor[3].blue  =     0;

        xcolor[4].flags = DoRed | DoGreen | DoBlue;
        xcolor[4].pixel = color_mapping[4];
        xcolor[4].red   =     0;
        xcolor[4].green =     0;
        xcolor[4].blue  = 65535;

        for (i=5; i<128; i++) {
           xcolor[i].flags = DoRed | DoGreen | DoBlue;
           xcolor[i].pixel = color_mapping[i];
           xcolor[i].red   = 0;
           xcolor[i].green = (i - 5) * 537;
           xcolor[i].blue  = 0;
        }

        segment.offset = 5;
        segment.length = 123;

        XStoreColors (display, default_colormap, xcolor, 128);
        XSync (display, FALSE);
     }

/*
 *  Open xgl.
 */
     sys_state = xgl_open (XGL_SYS_ST_ERROR_DETECTION, FALSE, NULL);

/*
 *  Exit if unable to open xgl.
 */
```

```
    if (sys_state == NULL) {
       printf ("Error:  Failed to open XGL. Exiting\n");
       exit (0);
    }

/*
 *  Get the xgl version number.
 */
    xgl_object_get (sys_state, XGL_SYS_ST_VERSION, &version);

/*
 *  Print the xgl version number.
 */
    printf ("\n");
    printf ("XGL Version = %s", version);

/*
 *  Print the color mode.
 */
    if (truecolor)
       printf ("Color mode = TrueColor\n");
    else
       printf ("Color mode = Index\n");

/*
 *  Set up the xgl descriptor structures for the XGL window
 *  rasters for the underlay and overlay windows.
 */
    ul_x_win.X_display = ul_display;
    ul_x_win.X_window  = ul_window;
    ul_x_win.X_screen  = DefaultScreen (ul_display);
    ul_desc.win_ras.type = XGL_WIN_X | XGL_WIN_X_PROTO_DEFAULT;
    ul_desc.win_ras.desc = &ul_x_win;

    ol_x_win.X_display = display;
    ol_x_win.X_window  = ol_window;
    ol_x_win.X_screen  = DefaultScreen (display);
    ol_desc.win_ras.type = XGL_WIN_X | XGL_WIN_X_PROTO_DEFAULT;
    ol_desc.win_ras.desc = &ol_x_win;

/*
 *  Define command line options that allow the user to specify
 *  xlib rendering for the underlay or overlay window or both.
 *  Note that these options may not work for all frame buffers.
 */
```

```
    {
      int             i, n;
      char           *s;
      for (i = 1; i < argc; i++) {
             s = argv[i];
             n = strlen(s);
             if (!strncmp("-ovlxlib", s, n)) {
             ol_desc.win_ras.type =
                 XGL_WIN_X | XGL_WIN_X_PROTO_XLIB;
             } else if (!strncmp("-ulxlib", s, n)) {
                 ul_desc.win_ras.type =
                 XGL_WIN_X | XGL_WIN_X_PROTO_XLIB;
             }
      }
    }

/*
 *  Map the windows.
 */
    XSetWMColormapWindows(ul_display,
           XtWindowOfObject(popup_shell2), &ol_window, 1);
    XMapWindow (ul_display, ul_window);
    XMapWindow (display, ol_window);
    do {
        XNextEvent (ul_display, &event);
    } while (event.type != Expose);

/*
 *  Select input.
 */
    XSelectInput (ul_display, ul_window, ExposureMask |
                                         ButtonPressMask |
                                         ButtonReleaseMask |
                                         ButtonMotionMask |
                                         StructureNotifyMask);

/*
 *  Create the xgl colormap object used by the overlay raster.
 */
   ol_cmap = xgl_object_create (sys_state, XGL_CMAP, NULL,
                                XGL_CMAP_NAME, ol_colormap,
                                XGL_CMAP_COLOR_TABLE_SIZE, 5,
                                NULL);

/*
 *  Create the xgl colormap object to be used by the underlay
```

```
 *  raster when the raster is 8 bit.
 */
   if (!truecolor) {
       cmap = xgl_object_create (sys_state, XGL_CMAP, NULL,
                                 XGL_CMAP_NAME, default_colormap,
                                 XGL_CMAP_COLOR_TABLE_SIZE, 128,
                                 NULL);
   }

/*
 *  Create the underlay raster.
 */
   if (truecolor) {
      raster = xgl_object_create (sys_state, XGL_WIN_RAS, &ul_desc,
                                 XGL_DEV_COLOR_TYPE, XGL_COLOR_RGB,
                                 XGL_WIN_RAS_BUFFERS_REQUESTED, 2,
                                 XGL_WIN_RAS_BUF_DRAW,    1,
                                 XGL_WIN_RAS_BUF_DISPLAY, 0,
                                 NULL);

      dir_light_color.rgb.r = 1.0;
      dir_light_color.rgb.g = 1.0;
      dir_light_color.rgb.b = 1.0;

      amb_light_color.rgb.r = 1.0;
      amb_light_color.rgb.g = 1.0;
      amb_light_color.rgb.b = 1.0;

      specular_color.rgb.r = 1.0;
      specular_color.rgb.g = 1.0;
      specular_color.rgb.b = 1.0;
   }
   else {
      raster = xgl_object_create (sys_state, XGL_WIN_RAS, &ul_desc,
                              XGL_DEV_COLOR_TYPE, XGL_COLOR_INDEX,
                              XGL_DEV_COLOR_MAP, cmap,
                             XGL_WIN_RAS_PIXEL_MAPPING, color_mapping,
                              XGL_WIN_RAS_BUFFERS_REQUESTED, 2,
                              XGL_WIN_RAS_BUF_DRAW,    1,
                              XGL_WIN_RAS_BUF_DISPLAY, 0,
                              NULL);

      dir_light_color.gray = 1.0;
      amb_light_color.gray = 1.0;
      specular_color.index = 0;
```

```
        xgl_object_set (cmap,
                        XGL_CMAP_RAMP_NUM, 1,
                        XGL_CMAP_RAMP_LIST, &segment,
                        NULL);
     }

  /*
   *  Create the overlay raster.
   */
     ol_raster = xgl_object_create (sys_state, XGL_WIN_RAS, &ol_desc,
                         XGL_DEV_COLOR_TYPE, XGL_COLOR_INDEX,
                         XGL_DEV_COLOR_MAP, ol_cmap,
                         XGL_WIN_RAS_PIXEL_MAPPING, ol_color_mapping,
                         NULL);

     if (ol_raster == NULL) {
        printf ("Cannot create overlay window raster\n");
        exit (0);
     }

  /*
   *  Create the xgl contexts.
   */
      ul_ctx = xgl_object_create (sys_state,XGL_3D_CTX, NULL,
               XGL_CTX_DEVICE, raster,
               XGL_CTX_DEFERRAL_MODE, XGL_DEFER_ASTI,
               XGL_CTX_VDC_WINDOW, &vdc_window,
               XGL_CTX_VDC_ORIENTATION, XGL_Y_UP_Z_TOWARD,
               XGL_CTX_VDC_MAP, XGL_VDC_MAP_ALL,
               XGL_CTX_ATEXT_CHAR_HEIGHT, 0.5,
               XGL_3D_CTX_SURF_FRONT_LIGHT_COMPONENT,
               XGL_LIGHT_ENABLE_COMP_AMBIENT    |
               XGL_LIGHT_ENABLE_COMP_SPECULAR   |
               XGL_LIGHT_ENABLE_COMP_DIFFUSE,
               XGL_3D_CTX_SURF_FRONT_AMBIENT,  0.30,
               XGL_3D_CTX_SURF_FRONT_DIFFUSE,  1.00,
               XGL_3D_CTX_SURF_FRONT_SPECULAR, 1.00,
               XGL_3D_CTX_SURF_FRONT_SPECULAR_POWER, 32.0,
               XGL_3D_CTX_SURF_FRONT_ILLUMINATION,
                            XGL_ILLUM_PER_FACET,
               XGL_3D_CTX_HLHSR_MODE, XGL_HLHSR_NONE,
               XGL_CTX_NEW_FRAME_ACTION,
                        XGL_CTX_NEW_FRAME_CLEAR |
                        XGL_CTX_NEW_FRAME_PAINT_TYPE_ACTION |
                        XGL_CTX_NEW_FRAME_SWITCH_BUFFER,
               XGL_3D_CTX_LIGHT_SWITCHES, light_switches,
```

```
                 XGL_3D_CTX_SURF_FRONT_SPECULAR_COLOR, &specular_color,
                 XGL_3D_CTX_LIGHT_NUM, 3,
                 XGL_3D_CTX_SURF_FACE_CULL, XGL_CULL_BACK,
                 XGL_CTX_BACKGROUND_COLOR, &ul_black,
                 NULL);

    if (!truecolor){
       surf_color.index = 127;
       xgl_object_set (ul_ctx, XGL_CTX_SURF_FRONT_COLOR,
                      &surf_color, NULL);
    }

/*  Note that overlay window clears to a transparent
 *  background by default.
 */
    ol_ctx = xgl_object_create (sys_state, XGL_3D_CTX, NULL,
                 XGL_CTX_DEVICE, ol_raster,
                 XGL_CTX_DEFERRAL_MODE, XGL_DEFER_ASTI,
                 XGL_CTX_VDC_WINDOW, &vdc_window,
                 XGL_CTX_VDC_ORIENTATION, XGL_Y_UP_Z_TOWARD,
                 XGL_CTX_VDC_MAP, XGL_VDC_MAP_ALL,
                 XGL_CTX_ATEXT_CHAR_HEIGHT, 0.05,
                 XGL_3D_CTX_SURF_FRONT_ILLUMINATION, XGL_ILLUM_NONE,
                 XGL_3D_CTX_HLHSR_MODE, XGL_HLHSR_NONE,
                 XGL_CTX_PAINT_TYPE, XGL_PAINT_OPAQUE,
                 XGL_CTX_STEXT_COLOR, &ol_white,
                 XGL_CTX_MARKER_COLOR, &ol_white,
                 XGL_CTX_LINE_COLOR, &ol_white,
                 XGL_CTX_SURF_FRONT_COLOR, &ol_white,
                 XGL_CTX_NEW_FRAME_ACTION, XGL_CTX_NEW_FRAME_CLEAR,
                 XGL_CTX_SURF_FRONT_COLOR_SELECTOR,
                             XGL_SURF_COLOR_CONTEXT,
                 NULL);

/*
 *  Get the light objects.
 */
   xgl_object_get (ul_ctx, XGL_3D_CTX_LIGHTS, lights);

/*
 *  Define the lights.
 */
   xgl_object_set (lights[0],
                   XGL_LIGHT_TYPE, XGL_LIGHT_AMBIENT,
                   XGL_LIGHT_COLOR, &amb_light_color,
                   NULL);
```

```
   xgl_object_set (lights[1],
                   XGL_LIGHT_TYPE, XGL_LIGHT_DIRECTIONAL,
                   XGL_LIGHT_DIRECTION, &light_dir,
                   XGL_LIGHT_COLOR, &dir_light_color,
                   NULL);

   xgl_object_set (lights[2],
                   XGL_LIGHT_TYPE, XGL_LIGHT_DIRECTIONAL,
                   XGL_LIGHT_DIRECTION, &light_dir2,
                   XGL_LIGHT_COLOR, &dir_light_color,
                   NULL);

/*
 *  Clear the screen.
 */
    xgl_context_new_frame (ul_ctx);

/*
 *  Inquire about XGL attributes.
 */
    inq = xgl_inquire (sys_state, &ul_desc);

/*
 *  Check for DGA.
 */
    if (inq->dga_flag) {
       printf ("DGA = YES\n");
    }
    else {
       printf ("DGA = NO\n");
    }

/*
 *  Print the name of the frame buffer.
 */
    printf ("Frame Buffer = %s\n", inq->name);

/*
 *  Print the number of buffers allocated.
 */
    xgl_object_get (raster, XGL_WIN_RAS_BUFFERS_ALLOCATED,
                         &buffers);
    printf ("Buffers = %d\n", buffers);
    printf ("\n");
```

```
/*
 *  Initialize the data.
 */
    data_init ();

/*
 *  Get the global model transform.
 */
    xgl_object_get (ul_ctx, XGL_CTX_GLOBAL_MODEL_TRANS,
                   &global_model_trans);

/*
 *  Render the object.
 */
    ul_traverse ();
    ul_traverse ();

/*
 *  Handle the events.
 *  ...
 *  See the example program transparency_ovl.c for details.
 */


}
```

For the subroutines and button callbacks in the remainder of the program, see the program transparency_ovl.c in $XGLHOME/demo/examples.

Color 5

This chapter discusses the XGL Color Models and Color Map object, and includes information on the following topics:

- Indexed and RGB Color
- XGL color pipeline
- Color map double buffering
- XGL and the Kodak Color Management System

This chapter provides examples using a color table, a color ramp, a color cube, and color map double buffering

Introduction to XGL Color

The XGL color scheme is built around the fundamental concept that XGL Rasters represent underlying hardware display devices. XGL Raster objects, therefore, are closely tied to the capabilities of the hardware. XGL Raster objects help free the application programmer from dealing with the hardware directly while providing a consistent interface for performing rendering operations. Because XGL Raster objects were designed to parallel the hardware, Rasters support the two most often used color models: *indexed* and *RGB*.

Indexed Color Space

The indexed color model uses a *color lookup table* to map index values to corresponding RGB values. A lookup table is an array containing a finite number of RGB colors or gray shades for display on the hardware. Each pixel to be rendered on the display device has an associated index. Before the pixel is displayed, the color for the pixel is retrieved from the lookup table using the index as a pointer to the appropriate values. Indexed color is known as *indirect* color because the displayed color is determined indirectly through a color lookup table.

RGB Color Space

RGB does not require a lookup table. The individual pixel's color is specified as a fractional component of each of the three monitor primary colors: red, green, and blue. Each component is a number between 0.0 and 1.0, and represents the additive weight the specific primary color contributes to the final color. Because the component values directly specify pixel color to the hardware, RGB is also known as *direct* color.

XGL Color Pipeline

The XGL color pipeline defines how colors are moved from the application's color space (or color type) to the device color type. The application's color type is defined with respect to the XGL Raster Device object attached to the current XGL Context. The color type for Rasters is set at Raster creation time. The hardware color type is fixed to the device characteristics, and is unchangeable.

If the color type requested by the application for the Raster is the same as the color type of the underlying hardware, no conversion is necessary. The color values of the pixels to be displayed are passed to the hardware without modification.

If the application's color type is different from the hardware color type, a color conversion is required. The conversion can take place before or after XGL performs the rendering calculations, depending on the capabilities of the underlying hardware. Color conversion for Memory Rasters is always done before rendering.

Two cases of color type conversion are possible. In one case, the application can request RGB colors when the underlying hardware is indexed. In this case, the RGB-to-indexed conversion is done using a *color cube* stored in an XGL Color Map object. In the second case of color type conversion, the application requests indexed colors, and the underlying hardware is RGB. In this case, indexed-to-RGB conversion is required. The conversion is carried out using a color table stored in the XGL Color Map object.

The XGL Context color type must be the same as the color type of its associated Raster. Therefore, only one color type conversion is required when going from application color space to hardware color space. Figure 5-1 illustrates the two types of color conversion.

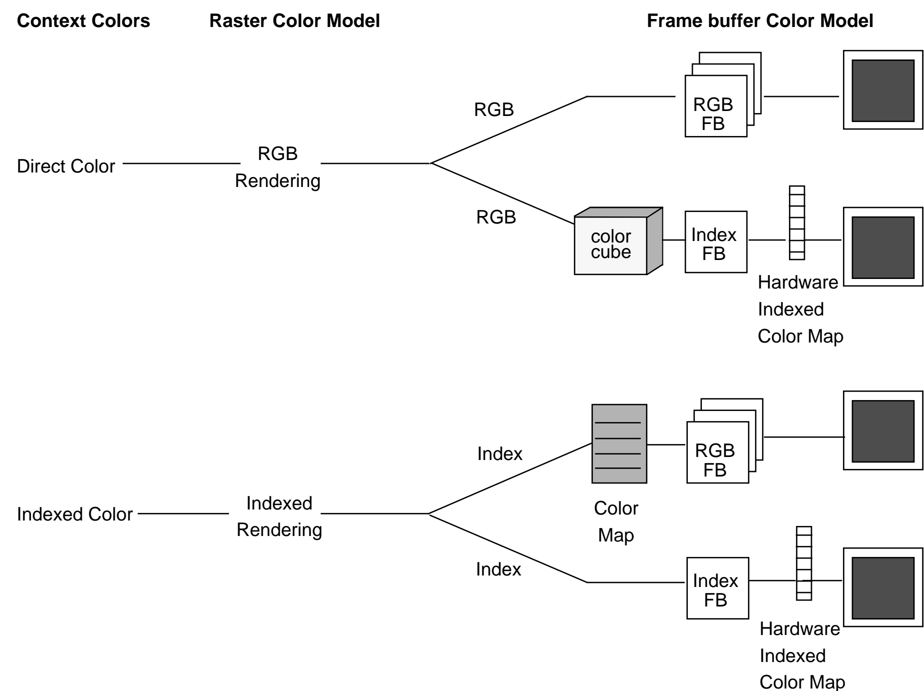


Figure 5-1 XGL Color Pipeline

Introduction to the Color Map Object

The XGL Color Map object permits an application programmer to add color to an XGL application. It defines color value conversions from one color type to another. Conversions are performed using either a color lookup table or a mapper function attached to the Color Map object. The color lookup table is normally used for indexed-to-RGB conversions, and the mapper function for RGB-to-indexed conversions.

Indexed Rasters must have a Color Map object associated with them. When the application creates an Raster object for an indexed raster, XGL attaches a default Color Map to the Raster if the application has not specified an application-created Color Map for the Raster.

RGB Rasters require a Color Map object only if the underlying display hardware is indexed. XGL attaches a default Color Map to an RGB Raster object only when the underlying hardware is indexed and the application program does not supply a Color Map.

Note – When using an indexed 8-bit Memory Raster to copy into an indexed 24-bit Window Raster, a separate Color Map object is required for each raster. Since XGL does internal color map conversion, which is different in these cases, the same color map cannot be used when the underlying hardware is different.

Creating a Color Map Object

The operator `xgl_object_create()`, with `XGL_CMAP` as the value for the parameter `type`, creates an XGL Color Map object. The `desc` parameter is ignored and can be set to `NULL`. As with all XGL objects, Color Map attributes can be set within the create call or with a separate `xgl_object_set()` command. The following code line shows Color Map object creation:

```
new_cmap = xgl_object_create(sys_st, XGL_CMAP, NULL, NULL);
```

Color Map objects are associated only with Raster and Metafile objects. A Color Map object is associated with a Device object with the XGL_DEV_COLOR_MAP attribute. This association can be set with an xgl_object_set() command, as in the following code line:

```
ras = xgl_object_set(ras, XGL_DEV_COLOR_MAP, new_cmap, NULL);
```

The two objects can also associated when the Raster is created, as in the following example:

```
xgl_object_create(sys_st, XGL_WIN_RAS, &obj_desc,
                  XGL_DEV_COLOR_TYPE, XGL_COLOR_INDEX,
                  XGL_DEV_COLOR_MAP, new_cmap,
                  NULL);
```

Color Tables

Most XGL applications exclusively use indexed Rasters, because most current hardware frame buffers are indexed. For indexed Rasters, the color table associated with the Raster Color Map object maps colors from the indexed frame buffer to the display device. The default color table, initialized at the creation of the Raster Color Map object, is a two-element, black-and-white color table. The following sections emphasize the use of indexed Rasters and the color table.

Color Table Attributes

The XGL Color Map object includes several attributes specific to color tables. The attribute XGL_CMAP_COLOR_TABLE specifies the color table that is to be attached to the Color Map object. XGL_CMAP_COLOR_TABLE_SIZE specifies the size of the color table and must be set prior to setting the color map with XGL_CMAP_COLOR_TABLE. The value of XGL_CMAP_COLOR_TABLE_SIZE should be greater than or equal to the sum of the start index and length structure elements passed with XGL_CMAP_COLOR_TABLE. Generally, the XGL_CMAP_COLOR_TABLE_SIZE attribute is set to a value that is a power of 2. If it exceeds the maximum size allowed by the hardware (as specified by XGL_CMAP_MAX_COLOR_TABLE_SIZE), then XGL sets this attribute to that maximum value.

Creating a Color Table

An XGL color table is formatted as a C structure of type *Xgl_color_list* containing a starting index of the color table, a color table length, and an array of *Xgl_color* values that comprise the heart of the color table. The contents of the array are specified as RGB color values. The color table structure is defined as:

```
/* XGL color table structure */
typedef struct {
  Xgl_usgn32   start_index;
  Xgl_usgn32   length;
  Xgl_color    *colors;
} Xgl_color_list;
```

The following code fragment creates an XGL Color Map object and attaches a color table to it.

```
/* colors to use in color table */
Xgl_color_rgb       GOLD = {1.00, 0.20, 0.00};
Xgl_color_rgb       GOLDENROD =  {1.00, 0.36, 0.00};
Xgl_cmap            ex_cmap;            /* color map object */
Xgl_color_list      ex_cmap_info;       /* color table */
Xgl_color           x_color_table[2]; /* color table array */
int                 i = 0;              /* index into array */

/* create color map object */
GBL_cmap = xgl_object_create(sys_st, XGL_CMAP, NULL, NULL);

/* initialize color array */
GBL_color_table[i++].rgb = GOLD;
GBL_color_table[i].rgb = GOLDENROD;

/* initialize color table struct */
GBL_cmap_info.start_index = 0;
GBL_cmap_info.length = 2;
GBL_cmap_info.colors = ex_color_table;

/* attach color table to color map object */
xgl_object_set(GBL_cmap,
                XGL_CMAP_COLOR_TABLE_SIZE, 2,
                XGL_CMAP_COLOR_TABLE, &ex_cmap_info,
                NULL);
```

Note – For an RGB Raster on 8-bit (indexed) hardware, the `start_index` field in the *Xgl_color_list* data structure is ignored. XGL takes the three R, G, B sizes and constructs a color cube for the application, filling in all the color map fields as it sees fit. The only control the application has if over the three dimensions of the color cube itself. For more information on color cubes, see "Color Cubes" on page 130, and see the example program on page 135.

Simple Color Table Example

The following example, `color_simple.c`, shows the basic features of a Color Map object. The example illustrates how to create a color table containing eight elements and a Color Map object using that color table. It also illustrates how to draw a series of eight rectangles, each using a different color from the color table.

The code in Code Example 5-1 is a part of a complete program that includes `ex_utils.c` and `color_main.c`, both of which are listed in Appendix B. To compile this program, type `make color` in the example program directory. The output of this program can be seen on Plate 3a.

Code Example 5-1 Simple Color Example

```
/*
 * color_simple.c
 */

#include <xview/xview.h>
#include <xgl/xgl.h>

#include "ex.h"

extern Xgl_rect_i2d      recti2d[256];
extern Xgl_rect_list     rectlist;
extern Xgl_cmap          simplecmap;

#defineCMAPSIZE8
#define RECTSIZE56

void
color_simple (Xgl_object ctx)
{
```

```
Xgl_sgn32           i;
Xgl_ras             ras;   /* raster associated with ctx */
Xgl_color           ecolor, rgb_colors[8];

/* get raster object from context */
xgl_object_get (ctx, XGL_CTX_DEVICE, &ras);

/* Set 8 colors: black, red, green, blue, yellow, cyan,
 * magenta, white
 */
rgb_colors[0].rgb.r = 0.0;
rgb_colors[0].rgb.g = 0.0;
rgb_colors[0].rgb.b = 0.0;
rgb_colors[1].rgb.r = 1.0;
rgb_colors[1].rgb.g = 0.0;
rgb_colors[1].rgb.b = 0.0;
rgb_colors[2].rgb.r = 0.0;
rgb_colors[2].rgb.g = 1.0;
rgb_colors[2].rgb.b = 0.0;
rgb_colors[3].rgb.r = 0.0;
rgb_colors[3].rgb.g = 0.0;
rgb_colors[3].rgb.b = 1.0;
rgb_colors[4].rgb.r = 1.0;
rgb_colors[4].rgb.g = 1.0;
rgb_colors[4].rgb.b = 0.0;
rgb_colors[5].rgb.r = 0.0;
rgb_colors[5].rgb.g = 1.0;
rgb_colors[5].rgb.b = 1.0;
rgb_colors[6].rgb.r = 1.0;
rgb_colors[6].rgb.g = 0.0;
rgb_colors[6].rgb.b = 1.0;
rgb_colors[7].rgb.r = 1.0;
rgb_colors[7].rgb.g = 1.0;
rgb_colors[7].rgb.b = 1.0;

/*
 * if color map associated with raster isn't simplecmap
 * then put it into the raster
 */
if (ex_color_type == XGL_COLOR_INDEX && !simplecmap) {
    Xgl_color_list      clist;      /* color table list */

    /* set up color list */
    clist.start_index = 0;
    clist.length = 8;
    clist.colors = rgb_colors;
```

```
        /*
         * create simple color map object and set the
         * color table to our new colors
         */
        simplecmap = xgl_object_create (sys_st, XGL_CMAP, NULL,
                    XGL_CMAP_COLOR_TABLE_SIZE, 8,
                    XGL_CMAP_COLOR_TABLE, &clist,
                    NULL);
    }

    /*
     * turn on edges to show color entries better
     */
    ecolor = white_color;
    xgl_object_set (ctx, XGL_CTX_SURF_EDGE_FLAG, TRUE,
            XGL_CTX_EDGE_COLOR, &ecolor,
            NULL);

    if (ex_color_type == XGL_COLOR_INDEX) {
        /*
         * put simple color map into raster
         */
        xgl_object_set (ras, XGL_RAS_COLOR_MAP, simplecmap, NULL);
    }

    rectlist.num_rects = 1;
    rectlist.rect_type = XGL_MULTIRECT_I2D;
    rectlist.bbox = NULL;

    xgl_object_set (ctx, XGL_CTX_PLANE_MASK, -1,
                    XGL_CTX_VDC_MAP, XGL_VDC_MAP_DEVICE, NULL);

    xgl_context_new_frame (ctx);

    /* Draw rectangles to display color map */
    for (i = 0; i < CMAPSIZE; i++) {
       Xgl_color            rectcolor;
       recti2d[i].corner_min.x = i * RECTSIZE;
       recti2d[i].corner_min.y = 0;
       recti2d[i].corner_min.flag = 1;
       recti2d[i].corner_max.x = (i * RECTSIZE) + (RECTSIZE - 1);
       recti2d[i].corner_max.y = RECTSIZE;

       rectlist.rects.i2d = &recti2d[i];
```

```
         if (ex_color_type == XGL_COLOR_INDEX)
             rectcolor.index = i;
         else {
              rectcolor = rgb_colors[i];
        }

        xgl_object_set (ctx,
             XGL_CTX_SURF_FRONT_COLOR, &rectcolor,
             NULL);

        xgl_multirectangle (ctx, &rectlist);
    }

    xgl_object_set (ctx, XGL_CTX_SURF_EDGE_FLAG, FALSE, NULL);
}
```

Color Table Ramps

Color tables can have color *ramps* embedded in them. A color ramp is a section of the color table of a specific length used to store color values that change by intensity only. For example, a gray ramp could have 32 color values from dark gray to light gray in a monotonically increasing fashion.

Ramps limit the range of valid indices used for indexed lighting and shading, and prevent overflow or underflow of color values. If an index defining the vertex colors in an XGL primitive is in a ramp, the frame buffer color values produced by the lighting and shading are limited to fall within the minimum and maximum index values of the ramp. Ramps are also important in lighting, shading, and depth cueing in indexed rasters.

Creating Color Table Ramps

An XGL color ramp is specified by using the C structure *Xgl_segment*, which contains an offset into the color table where the ramp begins, and the length of the ramp. If more than one ramp exists in the color table, an array of *Xgl_segment* structures is required, with each member of the array defining each ramp.The *Xgl_segment* structure is defined as:

```
/* XGL structure for defining ramps in color table */
typedef struct {
  Xgl_usgn32   offset;
  xgl_usgn32   length;
} Xgl_segment;
```

The number of ramps in the color table is set using the Color Map attribute XGL_CMAP_RAMP_NUM, and a pointer to the array of *Xgl_segment* structures associated with the ramps must be specified by setting the attribute XGL_CMAP_RAMP_LIST.

The following code fragment shows the steps required to add ramps to an XGL Color Map object. The first step initializes the section of the color table associated with each ramp. Then the *Xgl_segment* structures are initialized. Finally, the Color Map is created.

```
Xgl_cmap          GBL_cmap;       /* Color Map object */
Xgl_color_list    GBL_cmap_info;  /* color table */
Xgl_color         GBL_color_table[128]; /* color table array */
int               i;              /* index into array */
Xgl_segment       segments[4];    /* color ramps */

/* initialize color array */
/* create red ramp */
for (i = 0; i < 16; i++) {
  GBL_color_table[i].rgb.r = ((float)(i) / 15.0);
  GBL_color_table[i].rgb.g = GBL_color_table[i].rgb.b = 0.0;
}

/* create green ramp */
for (i = 16; i < 32; i++) {
  GBL_color_table[i].rgb.g = ((float)(i - 16) / 15.0);
  GBL_color_table[i].rgb.r = GBL_color_table[i].rgb.b = 0.0;
}
```

```
/* create blue ramp */
for (i = 32; i < 64; i++) {
  GBL_color_table[i].rgb.b = ((float)(i - 32) / 31.0);
  GBL_color_table[i].rgb.r = GBL_color_table[i].rgb.g = 0.0;
}
/* create gray ramp */
for (i = 64; i < 128; i++) {
  GBL_color_table[i].rgb.r =
  GBL_color_table[i].rgb.g =
  GBL_color_table[i].rgb.b = ((float)(i - 64) / 63.0);
}

/* initialize ramp structure */
segments[0].offset = 0; /* red ramp */
segments[0].length = 16;
segments[1].offset = 16;/* green ramp */
segments[1].length = 16;
segments[2].offset = 32;/* blue ramp */
segments[2].length = 32;
segments[3].offset = 64;/* gray ramp */
segments[3].length = 64;

/* initialize color table struct */
GBL_cmap_info.start_index = 0;
GBL_cmap_info.length = 128;
GBL_cmap_info.colors = GBL_color_table;

/* create Color Map object and attach color table to it */
GBL_cmap = xgl_object_create(sys_st, XGL_CMAP, NULL,
                  XGL_CMAP_COLOR_TABLE_SIZE, 128,
                  XGL_CMAP_RAMP_NUM, 4,
                  XGL_CMAP_RAMP_LIST, segments,
                  XGL_CMAP_COLOR_TABLE, &GBL_cmap_info,
                  NULL);
```

Color Ramp Example

The following example, `color_ramp.c`, shows how color ramps can be created and used in an XGL application. The code in Code Example 5-2 is a part of a complete program that includes `ex_utils.c` and `color_main.c`, both of which are listed in Appendix B. To compile this program, type `make color` in the example program directory. The output of this program can be seen on Plate 3b.

Code Example 5-2 Color Ramp Example

```
/*
 * color_ramp.c
 */

#include <xview/xview.h>
#include <xgl/xgl.h>

#include "ex.h"

#define RAMP_SIZE16

extern Xgl_pt_list       pl_3d;
extern Xgl_object        rampcmap;

void
color_ramp (Xgl_object   ctx)
{
    Xgl_object            ras;    /* raster associated with ctx */
    Xgl_object            cmap;   /* color map associated with ras */
    Xgl_color             ln_color; /* vector color */
    Xgl_object            view_trans;

    /*
     * get raster object from context
     */
    xgl_object_get (ctx, XGL_CTX_DEVICE, &ras);

    if (ex_color_type == XGL_COLOR_INDEX) {

   /*
    * get color map object from raster
    */
   xgl_object_get (ras, XGL_RAS_COLOR_MAP, &cmap);

   /*
    * if color map associated with raster isn't rampcmap then put
    * it into the raster
    */
   if (cmap != rampcmap) {

      if (!rampcmap) {
       Xgl_sgn32          i;
```

```
    Xgl_color_list     clist;      /* color table list */
    Xgl_color          colors[RAMP_SIZE];/* list of colors */
    Xgl_segment        segment;    /* color ramp segment */

    /*
     * create color list
     */
    clist.start_index = 0;
    clist.length      = RAMP_SIZE;
    clist.colors      = colors;

    /*
     * gray ramp for depth cued vectors
     */
    for (i = 0; i < RAMP_SIZE; i++) {
        colors[i].rgb.r =
        colors[i].rgb.g =
       colors[i].rgb.b = ((float) i) / ((float)(RAMP_SIZE - 1));
    }

    /*
     * setup segment data
     */
    segment.offset = 0;
    segment.length = RAMP_SIZE;

    /*
     * create and set attributes for ramp color map
     */
    rampcmap = xgl_object_create(sys_st, XGL_CMAP, NULL,
                XGL_CMAP_COLOR_TABLE_SIZE, RAMP_SIZE,
                XGL_CMAP_COLOR_TABLE, &clist,
                XGL_CMAP_RAMP_NUM, 1,
                XGL_CMAP_RAMP_LIST, &segment,
                NULL);
    }

     /*
      * set color map in raster
      */
     xgl_object_set (ras, XGL_RAS_COLOR_MAP, rampcmap, NULL);
    }

  } else {/* RGB raster */
    printf("no ramps on RGB rasters\n");
  }
```

```
    /*
     * modify the viewing transform so we can see the cube
     */
    xgl_object_get (ctx, XGL_CTX_VIEW_TRANS, &view_trans);

    xgl_transform_rotate (view_trans, 0.3,
            XGL_AXIS_Z, XGL_TRANS_POSTCONCAT);

    xgl_transform_rotate (view_trans, 0.3,
            XGL_AXIS_Y, XGL_TRANS_POSTCONCAT);

    xgl_transform_rotate (view_trans, 0.3,
            XGL_AXIS_X, XGL_TRANS_POSTCONCAT);

    /*
     * render a depth cued multipolyline
     */
   xgl_object_set (ctx, XGL_CTX_VDC_MAP, XGL_VDC_MAP_DEVICE, NULL);
    xgl_context_new_frame (ctx);
   xgl_object_set (ctx, XGL_CTX_VDC_MAP, XGL_VDC_MAP_ASPECT, NULL);

    xgl_object_set (ctx, XGL_3D_CTX_DEPTH_CUE_MODE,
                   XGL_DEPTH_CUE_LINEAR, NULL);

    if (ex_color_type == XGL_COLOR_INDEX)
          /* set color to highest value in ramp */
          ln_color.index = RAMP_SIZE - 1;
    else
          ln_color = white_color;
    xgl_object_set (ctx, XGL_CTX_LINE_COLOR, &ln_color, NULL);

    xgl_multipolyline (ctx, NULL, 1, &pl_3d);

    /* display color table */
    color_show (ras, -1);

    /* restore viewing transformation */
    xgl_transform_identity (view_trans);

    /* turn off depth cueing */
    xgl_object_set (ctx, XGL_3D_CTX_DEPTH_CUE_MODE,
                   XGL_DEPTH_CUE_OFF, NULL);
}
```

Color Mapping

Another way to map colors from one color space to another uses the Color Map attributes XGL_CMAP_COLOR_MAPPER, and XGL_CMAP_INVERSE_COLOR_MAPPER. These attributes point to functions available to the application programmer for manipulating colors, although typical color operations can be handled without them. The attribute XGL_CMAP_COLOR_MAPPER specifies a function that maps the color values from the device color space set by the application (with XGL_DEV_COLOR_TYPE) to the underlying real color space (as defined by XGL_DEV_REAL_COLOR_TYPE). XGL_CMAP_INVERSE_COLOR_MAPPER specifies a function that maps the values from the underlying real color space (set with XGL_DEV_REAL_COLOR_TYPE) to the device color space set by the application (XGL_DEV_COLOR_TYPE). For information on writing an application-specific color mapper, see the XGL_CMAP_COLOR_MAPPER reference page in the *XGL Reference Manual*.

Color Cubes

In the RGB color model, XGL uses three colors: red, green, and blue, which are assigned to three perpendicular axes to form the color cube. With a red axis, a green axis, and a blue axis, the color cube can represent points in every possible color in three dimensions with three values. The RGB color cube is an additive color system where color is generated by mixing various colored light sources.

The corner of the cube where the axes originate is at 0,0,0, and the color is black. At the opposite corner, where red, green, and blue are at their brightest, the RGB values are 1,1,1, and the color is white. Figure 5-2 illustrates the RGB color cube.

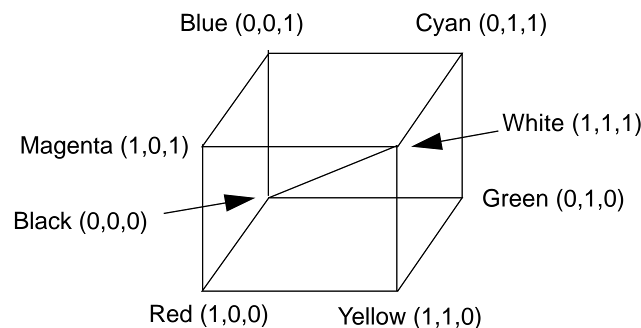


Figure 5-2 RGB Color Cube

The color cube is located in the color table associated with an XGL Color Map. The color cube converts RGB colors to indexed colors when the XGL Raster type is RGB and the underlying hardware is indexed. The attribute `XGL_CMAP_COLOR_CUBE_SIZE` sets the size of the color cube, specifying all three axes simultaneously. It is formatted as an array of three integers specifying the size of the red, green, and blue axes of the color cube, respectively. The default XGL color cube is red = 6, green = 9, and blue = 4. XGL is optimized for a color cube of this size. For an example of how a color cube is created, see the program on page 134.

Dithering

Dithering mixes two or more colored pixels to approximate a given color on the screen. This means placing pixels of different colors next to each other so that from a distance, the colors blend together to make a different color. When dithering in monochrome, the effect desired is a shade (or shades) of gray, with brightness depending on the ratio of black (off) to white (on) pixels. The number of shades available depends on the number of pixels comprising a *dither cell*. The more pixels in a dither cell, the more shades of gray available. More pixels per cell means lower resolution, however, as each shade of gray is approximated by 2×2 or 4×4 pixels, etc. In color dithering, the dither cell is filled with one or more colors defined in the current Color Map. When viewed together, the colors approximate the desired color.

If an application's Device color type is XGL_COLOR_RGB and the underlying hardware is indexed, XGL uses a color cube and dithering to convert the application's RGB colors to the display's indexed colors. The application can specify the dither mask that is required for the RGB-to-Index conversion using the attribute XGL_CMAP_DITHER_MASK. The read-only attribute XGL_CMAP_DITHER_MASK_N allows the application to get the dither mask size. The XGL Color Map dither mask must always be 8 x 8. To change the mask to 2×2 or 4×4, replicate the smaller mask into an 8×8 mask and then set it into the Color Map dither mask using the attribute XGL_CMAP_DITHER_MASK.

XGL does dithering automatically if the Device color type is RGB and the underlying hardware is indexed. For an example of color conversion from RGB to indexed colors and dithering, see the example program color_cube on page 135.

Sharing the Window System Color Map

The application can use Xlib function calls to create a color map that is used to map colors from the application color space to the hardware color space. This color map is the window system color map (not to be confused with the XGL Color Map object). The application passes the name of the window system color map to XGL so that XGL can share the use of the color map with the window system. Sharing the window system color map minimizes or prevents color map flashing.

To set up color map sharing, an application should do the following:

1. Create a window system color map using window system functions, such as the Xlib call XCreateColormap.
2. Get the color map name from the window system (the XID returned by the XCreateColormap call).
3. Get the pixel mapping array returned by XAllocColorCells, XAllocColor, XAllocColorPlanes, or XAllocNamedColor.
4. Set the XGL attribute XGL_CMAP_NAME to the window system color map name.

5. If the XGL_DEV_COLOR_TYPE attribute is XGL_COLOR_INDEX, set the color table size with XGL_CMAP_COLOR_TABLE_SIZE. If the XGL_DEV_COLOR_TYPE attribute is XGL_COLOR_RGB, set the size of the color cube with XGL_CMAP_COLOR_CUBE_SIZE.
6. Set XGL_WIN_RAS_PIXEL_MAPPING to the window system pixel mapping array. This tells the XGL application to use the pixel mapping from the window color map rather than from its own color map.

The application must ensure that the RGB values in the window system color map are correct. However, the application cannot query the contents of a window system produced color map using the XGL call xgl_object_get(). The application must manipulate the color map using window system calls.

If the application sets XGL_CMAP_NAME, XGL_WIN_RAS_PIXEL_MAPPING should be set immediately afterward to ensure that the application's colors are properly mapped to the hardware indices. XGL also alters the background color attribute (see XSetWindow attributes) when the application sets XGL_CMAP_NAME. The application should *not* change the value of the background pixel after XGL has set it.

The default value of XGL_CMAP_NAME is the XID of the X color map XGL uses to allocate the color cells. The value can be the same as the XID returned by the macro DefaultColormap, or it can be the XID of a private X color map, depending on the availability of read/write color cells in the default X color map.

When XGL creates its own Color Map (for example, the default Color Map object created when a new Window Raster object is created), XGL_CMAP_NAME reflects an internal X11 color map used by XGL and the window system. XGL implicitly uses the window system functions to create the X11 color map, associate it with an XGL Color Map object, and attach it to a Raster.

When allocating color cells from the window system, XGL does the following:

1. Attempts to allocate read/write cells from the default X color map.

   If this fails, XGL uses XCreateColormap to create a private X color map and uses XAllocColorCells to allocate the read/write color cells from it.

2. Converts the XGL color table associated with the color map object associated with the window raster object from RGB float values to XColor data structures.

3. XGL uses XStoreColors to update the read/write cells allocated in the first step.

Sharing the Window System Color Map Example

The following example program, color_cube.c, shows how an application can minimize color map flashing by sharing the window system color map. Color map flashing occurs on indexed frame buffers when an application overwrites the window system color map or another application's color map. The default color map, which is created when OpenWindows is invoked, has a fixed number of colors, for example 256 colors on an 8-bit frame buffer. This means that only 256 different colors can be allocated at one time. The window systm normally uses the first few entries of the default color map. If an XGL application uses the color entries set by the window system (or any other application), the server automatically switches the color map as the user moves the cursor over the XGL window, resulting in flashing of colors in other windows.

This example program shows color conversion from RGB to indexed colors using a color cube. When an XGL application sets XGL_DEV_COLOR_TYPE to XGL_COLOR_RGB and the underlying hardware is indexed, XGL uses a color cube to convert the application RGB colors to indexed colors. The default XGL color cube, as shown in this program, is red = 6, green = 9, and blue = 4, resulting in 216 colors. When XGL creates the color map and there are more than 128 colors in the color map, the XGL color map starts at index 0, thereby overwriting other color maps and causing color map flashing. Sharing the window system color map is a way to avoid this problem.

The program creates an RGB window raster on an 8-bit PseudoColor window. Using Xlib calls, the program creates a private X color map with a (6,9,4) color cube, The color cube pixels are offset by 32 in the color map. The program copies the first 32 colors from the default window system color map, defines the remaining colors for the color cube, and stores them into the private color map. It sets the XGL_CMAP_NAME attribute to the name of the private color map to inform XGL that the application will manage the X color map itself. The program then passes XGL the X-created color map and the array mapping the application color indices to the window system color indices. The program creates three faces of a color cube, rotates it, and renders. To quit the program, move the cursor on the window raster and press Return. Note that when the Raster color type is set to RGB on indexed hardware, dithering of colors is provided as part of the color conversion.

To compile the program, type `make color_cube`.

Code Example 5-3 Sharing the Window System Color Map

```
#include <stdio.h>
#include <math.h>
#include <X11/X.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>

#include <xgl/xgl.h>

static char     label[] = "XGL Colorcube";

#define RSIZE   6
#define GSIZE   9
#define BSIZE   4
#define NCOLORS RSIZE*GSIZE*BSIZE
#define OFFSET  32

Display          *display;
Window           window;
int              screen;

static Xgl_sys_state    sys_state;
static Xgl_ctx          ctx;
static Xgl_win_ras      ras;
static Xgl_cmap         cmap;
static int color_cube[] = {RSIZE, GSIZE, BSIZE};

static Xgl_pt_list      pl[3];
static Xgl_pt_color_f3d pts0[4], pts1[4], pts2[4];

Xgl_boolean
setup_colorcube(xcmap, pixel_mapping)
Colormap        *xcmap;
unsigned long   pixel_mapping[];
{
    int             i, j, r, g, b, rstep, gstep, bstep;
    int             ncells;
    unsigned long   xpmask[256];
    unsigned long   xpixels[256];
    XColor          xcolors[256];

    /* Create a private colormap */
```

```
*xcmap = XCreateColormap(display, window,
    DefaultVisual(display, screen), AllocNone);

/* Allocate the color cells in the private colormap */
ncells = OFFSET + NCOLORS;
if (!XAllocColorCells(display, *xcmap, True, xpmask, 0,
    xpixels, ncells)) {
    fprintf (stderr, "cannot allocate x color cells \n");
    exit(1);
}

/* Copy the first OFFSET colors from default color map
 * to the private color map
 */
for (i = 0; i < OFFSET; i++)
    xcolors[i].pixel = i;

XQueryColors(display, DefaultColormap(display, screen),
              xcolors, OFFSET);

/*
 * Set up the XColor data structure for the remaining colors and
 * set up the pixel mapping array to be passed back to XGL.
 * The pixel mapping array maps application color indices to
 * the window system color indices.
 */
for (i = 0; i < NCOLORS; i++) {
    j = OFFSET + i;
    xcolors[j].pixel = j;
    xcolors[j].flags = DoRed | DoGreen | DoBlue;
    pixel_mapping[i] = xcolors[j].pixel;
}

/* Set colors for the color cube */
rstep = 65535 / (RSIZE - 1);
gstep = 65535 / (GSIZE - 1);
bstep = 65535 / (BSIZE - 1);


for (b = 0; b < BSIZE; b++) {
    for (g = 0; g < GSIZE; g++) {
        for (r = 0; r < RSIZE; r++) {
            i = OFFSET + r + (RSIZE * ( g + GSIZE * b));
            xcolors[i].red = r * rstep;
            xcolors[i].green = g * gstep;
            xcolors[i].blue = b * bstep;
```

```
            }
        }
    }

    /* Store the RGB values of the color cells into
    /* the private colormap
    /*
    XStoreColors(display, *xcmap, xcolors, ncells);

    return TRUE;
}

/* Define points for the three faces of a cube */

void init_pts()
{
    /* first face */
    pl[0].pt_type = XGL_PT_COLOR_F3D;
    pl[0].bbox = NULL;
    pl[0].num_pts = 4;
    pl[0].num_data_values = 0;
    pl[0].pts.color_f3d = pts0;

    pts0[0].x = 0.75;
    pts0[0].y = 0.0;
    pts0[0].z = 0.0;
    pts0[0].color.rgb.r = 1.0;
    pts0[0].color.rgb.g = 0.0;
    pts0[0].color.rgb.b = 0.0;

    pts0[1].x = 0.75;
    pts0[1].y = 0.75;
    pts0[1].z = 0.0;
    pts0[1].color.rgb.r = 1.0;
    pts0[1].color.rgb.g = 1.0;
    pts0[1].color.rgb.b = 0.0;

    pts0[2].x = 0.75;
    pts0[2].y = 0.75;
    pts0[2].z = 0.75;
    pts0[2].color.rgb.r = 1.0;
    pts0[2].color.rgb.g = 1.0;
    pts0[2].color.rgb.b = 1.0;

    pts0[3].x = 0.75;
    pts0[3].y = 0.0;
```

```
pts0[3].z = 0.75;
pts0[3].color.rgb.r = 1.0;
pts0[3].color.rgb.g = 0.0;
pts0[3].color.rgb.b = 1.0;

/* second face */
pl[1].pt_type = XGL_PT_COLOR_F3D;
pl[1].bbox = NULL;
pl[1].num_pts = 4;
pl[1].num_data_values = 0;
pl[1].pts.color_f3d = pts1;

pts1[0].x = 0.75;
pts1[0].y = 0.75;
pts1[0].z = 0.0;
pts1[0].color.rgb.r = 1.0;
pts1[0].color.rgb.g = 1.0;
pts1[0].color.rgb.b = 0.0;

pts1[1].x = 0.0;
pts1[1].y = 0.75;
pts1[1].z = 0.0;
pts1[1].color.rgb.r = 0.0;
pts1[1].color.rgb.g = 1.0;
pts1[1].color.rgb.b = 0.0;

pts1[2].x = 0.0;
pts1[2].y = 0.75;
pts1[2].z = 0.75;
pts1[2].color.rgb.r = 0.0;
pts1[2].color.rgb.g = 1.0;
pts1[2].color.rgb.b = 1.0;

pts1[3].x = 0.75;
pts1[3].y = 0.75;
pts1[3].z = 0.75;
pts1[3].color.rgb.r = 1.0;
pts1[3].color.rgb.g = 1.0;
pts1[3].color.rgb.b = 1.0;

/* third face */
pl[2].pt_type = XGL_PT_COLOR_F3D;
pl[2].bbox = NULL;
pl[2].num_pts = 4;
pl[2].num_data_values = 0;
pl[2].pts.color_f3d = pts2;
```

```
    pts2[0].x = 0.0;
    pts2[0].y = 0.0;
    pts2[0].z = 0.75;
    pts2[0].color.rgb.r = 0.0;
    pts2[0].color.rgb.g = 0.0;
    pts2[0].color.rgb.b = 1.0;

    pts2[1].x = 0.75;
    pts2[1].y = 0.0;
    pts2[1].z = 0.75;
    pts2[1].color.rgb.r = 1.0;
    pts2[1].color.rgb.g = 0.0;
    pts2[1].color.rgb.b = 1.0;

    pts2[2].x = 0.75;
    pts2[2].y = 0.75;
    pts2[2].z = 0.75;
    pts2[2].color.rgb.r = 1.0;
    pts2[2].color.rgb.g = 1.0;
    pts2[2].color.rgb.b = 1.0;

    pts2[3].x = 0.0;
    pts2[3].y = 0.75;
    pts2[3].z = 0.75;
    pts2[3].color.rgb.r = 0.0;
    pts2[3].color.rgb.g = 1.0;
    pts2[3].color.rgb.b = 1.0;
}

/* Rotates the color cube to the appropriate view */

void setup_view()
{
    Xgl_trans   view_trans;

    xgl_object_get(ctx, XGL_CTX_VIEW_TRANS, &view_trans);

    /* swing the y-axis of VDC so that it is vertical in WC */
    xgl_transform_rotate(view_trans, -M_PI/2, XGL_AXIS_X,
                         XGL_TRANS_POSTCONCAT);

    /* swivel about the vertical axis */
    xgl_transform_rotate(view_trans, -3*M_PI/4, XGL_AXIS_Y,
                         XGL_TRANS_POSTCONCAT);
```

```
    /* tip VDC */
    xgl_transform_rotate(view_trans, M_PI/6, XGL_AXIS_X,
                         XGL_TRANS_POSTCONCAT);
}

/* Draw the color cube */

void draw()
{
    Xgl_pt_f3d      ref_pos, ann_pos;

    xgl_context_new_frame(ctx);

    xgl_multi_simple_polygon(ctx, XGL_FACET_FLAG_SIDES_ARE_4,
        NULL, NULL, 3, pl);

    ref_pos.x = 0.75;
    ref_pos.y = 0.0;
    ref_pos.z = -0.75;
    ann_pos.x = ann_pos.y = ann_pos.z = 0.0;

    xgl_annotation_text(ctx, "Hit <Return> to quit",
                         &ref_pos, &ann_pos);

    xgl_context_flush(ctx, XGL_FLUSH_BUFFERS);
}

void resize()
{
    Xgl_bounds_d3d  vdc_win, dc_vp;
    Xgl_pt_d3d      ras_max_coords;

    xgl_window_raster_resize(ras);

    xgl_object_get(ras, XGL_DEV_MAXIMUM_COORDINATES,
                      &ras_max_coords);

    dc_vp.xmin = 0.0;
    dc_vp.xmax = ras_max_coords.x;
    dc_vp.ymin = 0.0;
    dc_vp.ymax = ras_max_coords.y;
    dc_vp.zmin = 0.0;
    dc_vp.zmax = ras_max_coords.z;

    xgl_object_set(ctx, XGL_CTX_DC_VIEWPORT, &dc_vp, NULL);
}
```

```
main(argc, argv)
int argc;
char *argv[];
{
    XSetWindowAttributes attr;
    Xgl_X_window      x_win;
    XSizeHints        size_hints;
    XWMHints          hints ;
    Xgl_obj_desc      desc;
    XEvent            event;
    Colormap          x_cmap;
    unsigned long     pixel_mapping[NCOLORS];
    Xgl_bounds_d3d    bounds;
    Xgl_color         back_color, text_color;
    float             ann_cheight;

    if ((display = XOpenDisplay (NULL)) == NULL) {
        (void) fprintf (stderr, "cannot open display\n");
        exit (1);
    }
    screen  = DefaultScreen (display);

    if (DefaultDepth(display, screen) != 8) {
         fprintf(stderr, "default screen depth != 8\n");
         exit(1);
     }

    attr.colormap = DefaultColormap(display, screen);

    attr.event_mask = ButtonPressMask | ButtonReleaseMask |
           KeyPressMask |StructureNotifyMask | ExposureMask;

    window = XCreateWindow (display,
        RootWindow(display,screen),
        100, 100,
        500, 400, 0,
        8,
        InputOutput,
        DefaultVisual(display, screen),
        CWEventMask | CWColormap,
        &attr);

    size_hints.x = 100 ;
    size_hints.y = 100 ;
    size_hints.width = 500 ;
```

```
size_hints.height = 400 ;
size_hints.flags = PPosition | PSize ;

XSetStandardProperties(display, window, label, label,
    NULL, argv, argc, &size_hints );

XMapWindow (display, window);
do {
    XNextEvent (display, &event);
} while (event.type != Expose);

sys_state = xgl_open(NULL);

/* Create XGL RGB raster */
x_win.X_display = display;
x_win.X_window  = window;
x_win.X_screen = screen;
desc.win_ras.type = XGL_WIN_X;
desc.win_ras.desc = &x_win;

ras = (Xgl_object) xgl_object_create(sys_state,
    XGL_WIN_RAS, &desc,
    XGL_DEV_COLOR_TYPE, XGL_COLOR_RGB,
    NULL);

/* Create a private colormap for the color cube */
setup_colorcube(&x_cmap, pixel_mapping);

xgl_object_get(ras, XGL_DEV_COLOR_MAP, &cmap);

/* Pass XGL the name of the private color map */
xgl_object_set(cmap,
    XGL_CMAP_COLOR_CUBE_SIZE, color_cube,
    XGL_CMAP_NAME, x_cmap,
    NULL);

/*
 * Pass XGL the array mapping the application color indices to
 * the indices of the window system colors
 */
xgl_object_set(ras, XGL_WIN_RAS_PIXEL_MAPPING, pixel_mapping,
               NULL);

/* Attach the private color map to the window */
XSetWindowColormap(display, window, x_cmap);
XSync(display, False);
```

```
    bounds.xmin = bounds.ymin = -1.0;
    bounds.zmin = 0.0;
    bounds.xmax = bounds.ymax = 1.0;
    bounds.zmax = 2.0;

    back_color.rgb.r = back_color.rgb.g = back_color.rgb.b = 0.0;
    text_color.rgb.r = text_color.rgb.g = text_color.rgb.b = 1.0;
    ann_cheight = 0.05;

    /* Create the XGL Context */
    ctx = xgl_object_create(sys_state, XGL_3D_CTX, NULL,
        XGL_CTX_DEVICE, ras,
        XGL_CTX_VDC_ORIENTATION, XGL_Y_UP_Z_TOWARD,
        XGL_CTX_VDC_MAP, XGL_VDC_MAP_OFF,
        XGL_CTX_VDC_WINDOW, &bounds,
        XGL_3D_CTX_SURF_FRONT_ILLUMINATION,
                   XGL_ILLUM_NONE_INTERP_COLOR,
        XGL_CTX_BACKGROUND_COLOR, &back_color,
        XGL_CTX_STEXT_COLOR, &text_color,
        XGL_CTX_ATEXT_CHAR_HEIGHT, ann_cheight,
        XGL_3D_CTX_HLHSR_MODE, XGL_HLHSR_Z_BUFFER,
        XGL_CTX_NEW_FRAME_ACTION,
          XGL_CTX_NEW_FRAME_CLEAR | XGL_CTX_NEW_FRAME_HLHSR_ACTION,
        NULL);

    init_pts();
    setup_view();

    draw();

    while (1) {
        XNextEvent (display, &event);
        if (event.type == Expose) {
            draw();
        } else if (event.type == ButtonPress) {
            draw();
        } else if (event.type == ConfigureNotify) {
            resize();
        } else if (event.type == KeyPress) {
            xgl_close(sys_state);
            exit(0);
        }
    }
}
```

Tips on 8-Bit Indexed Color Map Management

XGL allocates color map entries by 8-bit color planes. For example, if your XGL_CMAP_COLOR_CUBE_SIZE multiplies to eight, XGL allocates eight color cell entries. If the size multiplies to nine, XGL allocates 16 (2^4) color cells. If the application requests 120 cells, XGL allocates 128 color entries even though the application doesn't use all the color cells. If the application requires 140 color cell entries (2^8 = 256 cells), XGL allocates 256 color cells, so the entire color map is private to the XGL application.

The color_cube.c example program requires at least 216 color map entires, so XGL uses the entire color map, and the program flashes if another winodw requires its own colors. If your XGL program requires fewer than 128 color map entries, you can avoid color map flashing by sharing the color map as shown in the color_cube.c OFFSET macro. If you use this approach to share eight window system entries (0 through 7) and use 120 entries for your XGL application, there will be 128 color map entries available for other applications or windows.

Color Map Double Buffering

Double buffering produces clean animation because rendering occurs in an hidden memory buffer while the contents of a second buffer are displayed. Once rendering is complete in the hidden buffer, its contents are switched with the contents of the buffer in view. The buffer that was in view is cleared, and a new scene is rendered into it. By quickly switching between buffers, an application can snap an image as a completely rendered scene into the display area. Rendering always occurs in the hidden buffer.

Some hardware has separate areas of memory specifically available for double buffering. Hardware that does not have such capabilities can use the plane enable mask in conjunction with XGL Color Maps for double buffering. The Raster's Color Map is toggled between two Color Maps working in tandem with two values for the Context's plane-enable mask. This is called *color map double buffering*.

Color Map Double Buffering Example

The following example, `color_dbuf.c`, uses color map double buffering to animate the rotation of a cube. The double buffering makes the motion smooth and visually appealing. In this example, both buffers use two pixel bit planes. The bit planes define the bits of the color value associated with a pixel that will be used when displaying color on the device. For example, in an 8-bit frame buffer, where eight bits comprise each color value, any or all of the eight bits (or eight bit planes) can be enabled for displaying on the device.

Color map double buffering is implemented using the plane enable mask and two color maps. Two bits are used for each buffer, which results in a total of four colors. Only two (2^2) bits are used to specify colors in each buffer, and only a small number of colors can be used.

The benefits of color map double buffering need to be weighed against the cost of limiting color usage. Buffer 1 uses bit planes 1 and 2 (the first two bits of the pixel color), and buffer 2 uses planes 3 and 4. These are specified by setting the XGL attribute `XGL_CTX_PLANE_MASK`.

Rendering into buffer 1 is done when buffer 2 is displayed, and vice versa. Thus, when rendering into buffer 1, the Context's plane enable mask (`XGL_CTX_PLANE_MASK`) is set to 0xc (selecting the bits associated with buffer 2), which causes buffer 2 to be displayed. The mask is set to 0x3 when buffer 1 is displayed.

The application's indexing scheme should know whether XGL is drawing into buffer 1 or buffer 2, and should produce the same colors in either buffer. The following convention works well: when specifying colors, use the same index value for the same color in both buffers. If color 2 is used in the first buffer, it should be set in the second buffer as well. Color 2 is described as the index (2) in buffer 1 or (2 << 3) in buffer 2.

The code in Code Example 5-4 is a part of a complete program. The complete program includes `ex_utils.c` and `color_main.c`, both of which are listed in Appendix B. To compile this program, type `make color` in the example program directory. See Plates 4a and 4b for an illustration of this example.

Code Example 5-4 Color Map Double Buffering Example

```
/*
 * color_dbuf.c
 */

#include <xview/xview.h>
#include <xgl/xgl.h>
#include "ex.h"

extern Xgl_pt_list    pl_3d;
extern Xgl_object     dbufcmap1, dbufcmap2;
static Xgl_sgn32      current_buffer_is_buffer_1 = 1;
static void           switch_buffer(Xgl_object, Xgl_object);

void
color_dbuf (Xgl_object   ctx)
{
    Xgl_sgn32             i;
    double                angle;
    Xgl_object            ras;    /* raster associated with ctx */
    Xgl_object            cmap;  /* color map associated with
                                  * ras */
    Xgl_color             color[4];
    Xgl_color             ln_color;
    Xgl_object            view_trans;

    /*
    * if we are on an indexed raster then create two color maps to do
     * color map double buffering
     */
    if (ex_color_type == XGL_COLOR_INDEX) {
        /* get raster object from context */
        xgl_object_get (ctx, XGL_CTX_DEVICE, &ras);

         /* get color map object from raster */
        xgl_object_get (ras, XGL_RAS_COLOR_MAP, &cmap);

        /*
         * if color map associated with raster isn't rampcmap then
         * put it into the raster
         */
        if (cmap != dbufcmap1 && cmap != dbufcmap2) {
          if (!dbufcmap1) {
            Xgl_color_list        clist;/* color table list */
            Xgl_color             colors[16];/* list of colors */
```

```
         /* create color list */
        clist.start_index = 0;
        clist.length = 16;
        clist.colors = colors;

        /*
         * Set color maps to have 4 colors per color map:
        * black, green,blue, white
         */
        for (i = 0; i < 16; i++) {
        switch (i & 3) {
        case 0:
            colors[i].rgb.r = 0.0;
            colors[i].rgb.g = 0.0;
            colors[i].rgb.b = 0.0;
            break;
        case 1:
            colors[i].rgb.r = 0.0;
            colors[i].rgb.g = 1.0;
            colors[i].rgb.b = 0.0;
            break;
        case 2:
            colors[i].rgb.r = 0.0;
            colors[i].rgb.g = 0.0;
            colors[i].rgb.b = 1.0;
            break;
        case 3:
            colors[i].rgb.r = 1.0;
            colors[i].rgb.g = 1.0;
            colors[i].rgb.b = 1.0;
            break;
          }
        }

        /*
         * create buffer 1 color map object and set
        * the color table to our new colors
         */
        dbufcmap1 = xgl_object_create (sys_st, XGL_CMAP, NULL,
                        XGL_CMAP_COLOR_TABLE_SIZE, 16,
                        XGL_CMAP_COLOR_TABLE, &clist,
                        NULL);
    }

    if (!dbufcmap2) {
```

```
Xgl_color_list       clist;/* color table list */
Xgl_color            colors[16];/* list of colors */

/ * create color list */
clist.start_index = 0;
clist.length = 16;
clist.colors = colors;

/*
 * Set color maps to have 4 colors per
* color map:  black, green,blue, white
 */
for (i = 0; i < 16; i++) {
  switch (i >> 2) {

  case 0:
     colors[i].rgb.r = 0.0;
     colors[i].rgb.g = 0.0;
     colors[i].rgb.b = 0.0;
     break;
  case 1:
     colors[i].rgb.r = 0.0;
     colors[i].rgb.g = 1.0;
     colors[i].rgb.b = 0.0;
     break;
  case 2:
     colors[i].rgb.r = 0.0;
     colors[i].rgb.g = 0.0;
     colors[i].rgb.b = 1.0;
     break;
  case 3:
     colors[i].rgb.r = 1.0;
     colors[i].rgb.g = 1.0;
     colors[i].rgb.b = 1.0;
     break;
  }
}
/*
 * create buffer 2 color map object and set
* the color table to our new colors
 */
dbufcmap2 = xgl_object_create (sys_st, XGL_CMAP, NULL,
                XGL_CMAP_COLOR_TABLE_SIZE, 16,
                XGL_CMAP_COLOR_TABLE, &clist,
                NULL);
}
```

```
            }

            color[0].index = 0;
            color[1].index = (1 << 2) | 1;
            color[2].index = (2 << 2) | 2;
            color[3].index = (3 << 2) | 3;

            ln_color = color[3];
            xgl_object_set (ctx, XGL_CTX_LINE_COLOR, &ln_color, NULL);
     }
     else {
          /* RGB raster so just set color; no double buffering */
            printf("no color map double buffering on RGB rasters\n");
            ln_color = white_color;
            xgl_object_set (ctx, XGL_CTX_LINE_COLOR, &ln_color, NULL);
     }

    xgl_object_set (ctx, XGL_CTX_VDC_MAP, XGL_VDC_MAP_ASPECT, NULL);

     / * modify the viewing transform so we can see the cube */
     xgl_object_get (ctx, XGL_CTX_VIEW_TRANS, &view_trans);

     i = 0;
     angle = 0.;
     while (angle <= 6.28) {

    switch_buffer (ctx, ras);
     /* clear raster after waiting for retrace */
    xgl_object_set (ctx, XGL_CTX_NEW_FRAME_ACTION,
             XGL_CTX_NEW_FRAME_VRETRACE | XGL_CTX_NEW_FRAME_CLEAR,
             NULL);
    xgl_object_set (ctx, XGL_CTX_VDC_MAP, XGL_VDC_MAP_DEVICE, NULL);
    xgl_context_new_frame (ctx);
    xgl_object_set (ctx, XGL_CTX_VDC_MAP, XGL_VDC_MAP_ASPECT, NULL);

    / * rotate for animation */
    xgl_transform_rotate (view_trans, angle,
                 XGL_AXIS_Z, XGL_TRANS_REPLACE);

    xgl_transform_rotate (view_trans, angle,
                 XGL_AXIS_Y, XGL_TRANS_POSTCONCAT);

    xgl_transform_rotate (view_trans, angle,
                 XGL_AXIS_X, XGL_TRANS_POSTCONCAT);

    xgl_multipolyline (ctx, NULL, 1, &pl_3d);
```

```
      angle += .01;
    }

     xgl_transform_identity (view_trans);
     if (ex_color_type == XGL_COLOR_INDEX)
        xgl_object_set (ctx, XGL_CTX_PLANE_MASK, 0xff, NULL);
     color_show (ras, -1);
}

static
void
switch_buffer (
    Xgl_object            ctx,
    Xgl_object            ras)
{
    if (ex_color_type == XGL_COLOR_INDEX) {
        if (current_buffer_is_buffer_1) {
          xgl_object_set (ras, XGL_RAS_COLOR_MAP, dbufcmap1, NULL);
           xgl_object_set (ctx, XGL_CTX_PLANE_MASK, 0xc, NULL);
        }
        else {
          xgl_object_set (ras, XGL_RAS_COLOR_MAP, dbufcmap2, NULL);
           xgl_object_set (ctx, XGL_CTX_PLANE_MASK, 0x3, NULL);
        }
        current_buffer_is_buffer_1 ^= TRUE;
    }
}
```

XGL and the Kodak Color Management System

An XGL application can use the Kodak Color Management System (KCMS) to correct colors between different devices, such as CRT's and printers. Because different technologies are used to produce colors on monitors and printers, the same set of RGB values may look different on a monitor than on the printed page. KCMS provides a set of library functions that corrects for the differences in color display between devices.

KCMS uses the profiles of devices to correct colors between devices. A profile is a device-specific file containing the color characteristics of a device. The information contained in the profile allows KCMS to convert input color data to the appropriate color-corrected output color data. KCMS connects the profiles for each device to form a complete color profile, which is used to

convert colors between devices. KCMS provides profiles for several scanners, printers, monitors, and for photoCD input. The devices for which KCMS provides associated profiles are listed in the KCMS release notes.

KCMS includes a C application interface that an XGL application can use to correct colors between devices. For example, an XGL application can use KCMS to correct the colors of an XGL image so that the colors appear the same when when printed on a color printer. The application uses KCMS function calls to connect the profiles for the color monitor and printer to create a complete profile describing the color transformation for the monitor and the printer. The application passes KCMS the complete color profile and the address of the XGL image to be corrected. KCMS converts the image's RGB values into the RGB values that will appear the same when printed.

Another way in which an XGL application might use KCMS is to correct a scanned image used to texture map XGL primitives. In this case, the application scans the image using a supported scanner and converts it to an XGL memory raster. KCMS corrects the memory raster using the profiles for the scanner and the monitor. The application can then use the corrected memory raster to texture map the XGL primitive.

Note – KCMS only works on RGB memory rasters. Indexed rasters contain an index into the device color table rather than the colors for each pixel. KCMS uses the RGB values of the pixels to correct the colors.

For more information on KCMS, see the *KCMS Application Developer's Guide* and the *KCMS Calibration Tool User Guide* in the Solaris SDK documentation, and the *KCMS CMM Developer's Guide* and *KCMS CMM Reference Manual* in the Solaris DDK documentation.

XGL and KCMS Example Code

The following code fragment shows how to use KCMS to correct the colors printed on a color printer so they appear the same as those displayed on the monitor. The code fragment creates an XGL window raster and memory raster. Because XGL does not provide a way to obtain the address of a window raster, KCMS cannot correct the contents of a window raster directly. In this example, the code copies the contents of the window raster to the memory raster using `xgl_context_copy_buffer()`, and KCMS then accesses the contents of the memory raster via the memory raster address. Once the image has been

corrected, it can be sent to a printing utility. The code fragment uses the monitor profile for a Sony Multiscan monitor and a printer profile for a Kodak XKS8300 printer. The program includes the KCMS header files `kcs.h` and `kcstypes.h`.

```
/*
 * Example program fragment loads in a printer profile and
 * monitor profile, then color corrects the image data.
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <memory.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <xgl/xgl.h>
#include "X11/Xlib.h"
#include "X11/Xutil.h"
#include "xi_lib.h"

#include "kcms/kcs.h"
#include "kcms/kcstypes.h"

#define RASTER_WIDTH  300
#define RASTER_HEIGHT 300

Xgl_sys_state sys_st;
Xgl_obj_desc desc;
Xgl_obj_desc descras;

static    Xgl_3d_ctx ctx,mem_ctx;

main(int argc, char **argv)
{
    KcsProfileDesc    printerDesc, monitorDesc;
    KcsProfileId      printerProfile, monitorProfile;
    KcsProfileId      profileSequence[2], completeProfile;
    KcsStatusId       status;
    KcsAttributeValueattrValue;
    KcsAttributeName  i;
    KcsOperationType  op = (KcsOpReverse+KcsContImage);
    KcsPixelLayout    pixelLayoutIn;
    u_long            failedProfileNum;
```

```
Xgl_win_ras         window_raster;
Xgl_color           pixel_color;
Xgl_X_window        dwin;
float               j, k;
Xgl_pt_i2d          pos;
Xgl_object          memory_raster;
Xgl_usgn32*         addr;
Xgl_usgn8*          mem8;
Display           *dpy;
Window            xwin;
int               screen;
Xgl_X_window      win;

create_xwin(&dpy,&xwin,&screen);
sys_st = xgl_open(XGL_SYS_ST_ERROR_DETECTION, TRUE, NULL);

win.X_display = dpy;
win.X_screen = screen;
win.X_window = xwin;

descras.win_ras.type = XGL_WIN_X;
descras.win_ras.desc = (void *)&win;

window_raster = xgl_object_create(sys_st,XGL_WIN_RAS,&descras,
                 XGL_DEV_COLOR_TYPE, XGL_COLOR_RGB, 0);

ctx = xgl_object_create(sys_st,XGL_3D_CTX,&desc,
    XGL_CTX_DEVICE,                 window_raster,
    XGL_CTX_VDC_MAP,                XGL_VDC_MAP_OFF,
    XGL_CTX_NEW_FRAME_ACTION,       XGL_CTX_NEW_FRAME_CLEAR |
                               XGL_CTX_NEW_FRAME_HLHSR_ACTION,
    XGL_3D_CTX_HLHSR_MODE,          XGL_HLHSR_Z_BUFFER,
    0);

memory_raster = xgl_object_create (sys_st, XGL_MEM_RAS, 0,
                            XGL_DEV_COLOR_TYPE, XGL_COLOR_RGB,
                                 XGL_RAS_DEPTH, 32,
                                 XGL_RAS_WIDTH, RASTER_WIDTH,
                                XGL_RAS_HEIGHT, RASTER_HEIGHT,
                                 NULL);

mem_ctx = xgl_object_create(sys_st,XGL_3D_CTX,&desc,
    XGL_CTX_DEVICE,                 memory_raster,
    XGL_CTX_VDC_MAP,                XGL_VDC_MAP_OFF,
    XGL_CTX_NEW_FRAME_ACTION,       XGL_CTX_NEW_FRAME_CLEAR |
                               XGL_CTX_NEW_FRAME_HLHSR_ACTION,
```

```
            XGL_3D_CTX_HLHSR_MODE,          XGL_HLHSR_Z_BUFFER,
            XGL_CTX_SURF_FRONT_FILL_STYLE,  XGL_SURF_FILL_SOLID,
            0);

    xgl_context_new_frame(ctx);

/* Draw into window raster */

/* Display graphics */

/* Copy  the window raster contents into the memory raster */
xgl_context_copy_buffer( mem_ctx, 0, 0, window_raster);

/* Load the monitor profile
   kcmsEKsony1604.mon is the name of the profile file for a Sony
   Multiscan 16 or 19 in monitor. A list of profiles and their
   associated devices are listed in the KCMS release notes */

    monitorDesc.type = KcsSolarisProfile;
    monitorDesc.desc.solarisFile.fileName = "kcmsEKsony1604.mon";
    monitorDesc.desc.solarisFile.hostName = NULL;
    monitorDesc.desc.solarisFile.oflag = O_RDONLY;
    monitorDesc.desc.solarisFile.mode = 0;

    status = KcsLoadProfile(&monitorProfile, &monitorDesc,
                          KcsLoadAllNow);
    if (status != KCS_SUCCESS) {
       fprintf(stderr,"MonitoKcsLoadProfile failed
               error = 0x%x\n", status);
       exit(1);
    }

/* Load the profile for a Kodak XKS8300 Printer */

    printerDesc.type = KcsSolarisProfile;
    printerDesc.desc.solarisFile.fileName = "kcmsEKxls83004.out";
    printerDesc.desc.solarisFile.hostName = NULL;
    printerDesc.desc.solarisFile.oflag = O_RDONLY;
    printerDesc.desc.solarisFile.mode = 0;

    status = KcsLoadProfile(&printerProfile, &printerDesc,
                          KcsLoadAllNow);
    if (status != KCS_SUCCESS) {
       fprintf(stderr,"printer KcsLoadProfile failed
               error = 0x%x\n", status);
       exit(1);
```

```
}

/* Combine moniter and printer profiles to create a complete
   profile */
profileSequence[0] = monitorProfile;
profileSequence[1] = printerProfile;
status = KcsConnectProfiles(&completeProfile, 2,
                         profileSequence, op,
                         &failedProfileNum);
if (status != KCS_SUCCESS) {
   fprintf(stderr, "MakeProfile failed in profile number %d\n",
           failedProfileNum);
   exit(1);
}

/* Get address of the memory raster */
xgl_object_get(memory_raster,
              XGL_MEM_RAS_MEMORY_ADDRESS, &addr);

/* in a 32bit XGL memory raster  bits 0-7 are ignored,
    bits 8-15 are for blue, bits 16-23 are for green, and
    bits 24-31 are for  red
  See the SDK entries for KcsPixelLayout and KcsComponent
  for more information*/

mem8 = (Xgl_usgn8 *)addr;

/* Set up pixel layout */

/* Red channel */
pixelLayoutIn.numOfComp = 3;
pixelLayoutIn.component[KcsRGB_R].addr = (char *)mem8+3;
pixelLayoutIn.component[KcsRGB_R].bitOffset = 0;
pixelLayoutIn.component[KcsRGB_R].compType= KcsCompUFixed;
pixelLayoutIn.component[KcsRGB_R].compDepth = 8;
pixelLayoutIn.component[KcsRGB_R].colOffset = 4;
pixelLayoutIn.component[KcsRGB_R].rowOffset = RASTER_WIDTH*4;
pixelLayoutIn.component[KcsRGB_R].maxRow = RASTER_HEIGHT;
pixelLayoutIn.component[KcsRGB_R].maxCol = RASTER_WIDTH;
pixelLayoutIn.component[KcsRGB_R].rangeStart = 0;
pixelLayoutIn.component[KcsRGB_R].rangeEnd= 255.0;

/* Green channel */
pixelLayoutIn.component[KcsRGB_G].addr = (char *)mem8+2;
pixelLayoutIn.component[KcsRGB_G].bitOffset = 0;
pixelLayoutIn.component[KcsRGB_G].compType= KcsCompUFixed;
```

```
    pixelLayoutIn.component[KcsRGB_G].compDepth = 8;
    pixelLayoutIn.component[KcsRGB_G].colOffset = 4;
    pixelLayoutIn.component[KcsRGB_G].rowOffset = RASTER_WIDTH*4;
    pixelLayoutIn.component[KcsRGB_G].maxRow = RASTER_HEIGHT;
    pixelLayoutIn.component[KcsRGB_G].maxCol = RASTER_WIDTH;
    pixelLayoutIn.component[KcsRGB_G].rangeStart = 0;
    pixelLayoutIn.component[KcsRGB_G].rangeEnd= 255.0;

    /* Blue channel */
    pixelLayoutIn.component[KcsRGB_B].addr = (char *)mem8+1;
    pixelLayoutIn.component[KcsRGB_B].bitOffset = 0;
    pixelLayoutIn.component[KcsRGB_B].compType= KcsCompUFixed;
    pixelLayoutIn.component[KcsRGB_B].compDepth = 8;
    pixelLayoutIn.component[KcsRGB_B].colOffset = 4;
    pixelLayoutIn.component[KcsRGB_B].rowOffset = RASTER_WIDTH*4;
    pixelLayoutIn.component[KcsRGB_B].maxRow = RASTER_HEIGHT;
    pixelLayoutIn.component[KcsRGB_B].maxCol = RASTER_WIDTH;
    pixelLayoutIn.component[KcsRGB_B].rangeStart = 0;
    pixelLayoutIn.component[KcsRGB_B].rangeEnd= 255.0;

/* Correct the memory raster */
    status = KcsEvaluate(completeProfile, op, &pixelLayoutIn,
                           &pixelLayoutIn);

    if( status == KCS_SUCCESS){

      /* Print the corrected memory raster */

    } else {
       printf("error from KcsEvaluate \n");
    }

    /* Free the profiles */
    KcsFreeProfile(completeProfile);
    KcsFreeProfile(printerProfile);
    KcsFreeProfile(monitorProfile);

    xgl_close(sys_st);

    exit(0);
}
```

This code fragment is a function that corrects XGL RGB colors. The colors are first converted to 8-bit integers, corrected by KCMS, and then converted back to floats. This is necessary because the default Color Management Module can only handle a `compType` of `KcsCompUFixed` with a maximum of 8 bits per channel.

```
correct_xgl_color ( Xgl_color *color, KcsProfileId  completeProfile,
                   KcsOperationType op)
{
 char                *addr;
 KcsPixelLayout      pixelLayoutIn;
 KcsStatusId         status;
 unsigned char       color_array[3];

/* convert rgb to 8-bit integers */
 color_array[0] = color->rgb.r * 255.0;
 color_array[1] = color->rgb.g * 255.0;
 color_array[2] = color->rgb.b * 255.0;

 addr = (char *)color_array;

    pixelLayoutIn.numOfComp = 3;
    pixelLayoutIn.component[KcsRGB_R].addr = addr;
    pixelLayoutIn.component[KcsRGB_R].bitOffset = 0;
    pixelLayoutIn.component[KcsRGB_R].compType= KcsCompUFixed;
    pixelLayoutIn.component[KcsRGB_R].compDepth = 8;
    pixelLayoutIn.component[KcsRGB_R].colOffset = 3;
    pixelLayoutIn.component[KcsRGB_R].rowOffset = 3;
    pixelLayoutIn.component[KcsRGB_R].maxRow = 1;
    pixelLayoutIn.component[KcsRGB_R].maxCol = 1;
    pixelLayoutIn.component[KcsRGB_R].rangeStart = 0;
    pixelLayoutIn.component[KcsRGB_R].rangeEnd= 255;

    pixelLayoutIn.component[KcsRGB_G].addr = addr+1;
    pixelLayoutIn.component[KcsRGB_G].bitOffset = 0;
    pixelLayoutIn.component[KcsRGB_G].compType= KcsCompUFixed;
    pixelLayoutIn.component[KcsRGB_G].compDepth = 8;
    pixelLayoutIn.component[KcsRGB_G].colOffset = 3;
    pixelLayoutIn.component[KcsRGB_G].rowOffset = 3;
    pixelLayoutIn.component[KcsRGB_G].maxRow = 1;
    pixelLayoutIn.component[KcsRGB_G].maxCol = 1;
    pixelLayoutIn.component[KcsRGB_G].rangeStart = 0;
    pixelLayoutIn.component[KcsRGB_G].rangeEnd= 255;

    pixelLayoutIn.component[KcsRGB_B].addr = addr+2;
```

```
    pixelLayoutIn.component[KcsRGB_B].bitOffset = 0;
    pixelLayoutIn.component[KcsRGB_B].compType= KcsCompUFixed;
    pixelLayoutIn.component[KcsRGB_B].compDepth = 8;
    pixelLayoutIn.component[KcsRGB_B].colOffset = 3;
    pixelLayoutIn.component[KcsRGB_B].rowOffset = 3;
    pixelLayoutIn.component[KcsRGB_B].maxRow = 1;
    pixelLayoutIn.component[KcsRGB_B].maxCol = 1;
    pixelLayoutIn.component[KcsRGB_B].rangeStart = 0;
    pixelLayoutIn.component[KcsRGB_B].rangeEnd= 255;

    status = KcsEvaluate(completeProfile, op, &pixelLayoutIn,
                                &pixelLayoutIn);

    if( status != KCS_SUCCESS ) {
       printf("error from KcsEvaluate 0x%x\n",status);
       return;
    }

   /* convert 8-bit rgb to floats */

    color->rgb.r = color_array[0];
    color->rgb.r = color->rgb.r/255.0;

    color->rgb.g = color_array[1];
    color->rgb.g = color->rgb.g/255.0;

    color->rgb.b = color_array[2];
    color->rgb.b = color->rgb.b/255.0;
}
```

Contexts 6

This chapter introduces the XGL Context object. The chapter includes information on the following topics:

- Context utility operators
- Context stacks
- Environment Context attributes

More information on the Context object is provided in Chapter 7, "Primitives and Graphics Context Attributes" on page 169.

Introduction to the XGL Context Object

The Context object is an abstraction of an idealized graphics renderer that produces images on a specified Device. The Context object controls all rendering actions directed to the associated XGL Device.

The Context object contains many operators that can be divided into two general groups: *utility operators*, discussed in this chapter, and *primitives*, discussed in Chapter 7, "Primitives and Graphics Context Attributes." The utility operators perform tasks such as clearing the Device associated with a Context. Primitives are the basic drawing operators available to an XGL application.

Context attributes maintain the state information required for XGL rendering operations. The Context attributes, like the operators, can be divided into two subsets, *graphics Context* attributes and *environment Context* attributes. The

graphics Context attributes directly affect the rendering of XGL primitives (for example, line color). They constitute the Context's ***graphics state***. The environment Context attributes are not directly associated with the way primitives are drawn on the Device. They constitute the Context's ***environment state***. Graphics state information is saved on an internal Context stack when the `xgl_context_push()` operator is invoked. Environment state information, on the other hand, cannot be pushed onto an internal stack.

Applications render graphics through the Context to a Device. Each Context is associated with a Device object that represents a display surface for all rendering operations. Although multiple Context objects can be simultaneously attached to a Device, each Context can only be associated with one Device object at a time.

This chapter discusses utility operators for Contexts, and describes environment attributes. The following chapter discusses the XGL primitive operators and introduces the graphics attributes. More detailed discussions of many of the Context graphics attributes can be found throughout this manual in sections describing the particular objects affected by the attributes.

Creating a Context Object

A Context object is created by the `xgl_object_create()` operator, using either `XGL_2D_CTX` or `XGL_3D_CTX` as the value for the parameter *type*. 2D or 3D Context attributes can be set within the create call or with a separate `xgl_object_set()` call.

```
ctx = xgl_object_create (sys_st, XGL_3D_CTX, NULL,
                          <attribute_list>,
                          NULL);
```

For rendering to occur, the application must explicitly associate a Device object with the Context object, either at Context creation time or later, using the `XGL_CTX_DEVICE` attribute and the handle to the Device object. A Device is not associated with the Context until this relationship is established.

The application can also associate the Context object with other objects that serve as resources containing information relevant for rendering. When XGL is initialized, the Context object is associated with ten Line Pattern objects containing the predefined line patterns, a Stroke Font object specifying the

default font, eight Marker objects containing the predefined marker definitions, and several Transform objects. As the application creates new objects, it must associate these objects with the Context object. This association can be made when the Context object is created or set later with the `xgl_object_set()` operator. Table 6-1 lists the objects that can be associated with the Context object and lists the linking attributes.

Table 6-1 Objects Associated with the Context Object

User Object	Used Object	Linking Attribute
2D Context and 3D Context	Window Raster Device Memory Raster Device Stream Device	XGL_CTX_DEVICE XGL_CTX_DEVICE XGL_CTX_DEVICE
	Transform	XGL_CTX_GLOBAL_MODEL_TRANS XGL_CTX_LOCAL_MODEL_TRANS XGL_CTX_MODEL_TRANS XGL_CTX_VIEW_TRANS XGL_CTX_MC_TO_DC_TRANS
	Line Pattern	XGL_CTX_LINE_PATTERN XGL_CTX_EDGE_PATTERN
	Marker	XGL_CTX_MARKER
	Memory Raster	XGL_CTX_RASTER_FPAT XGL_CTX_SURF_FRONT_FPAT
	Stroke Font	XGL_CTX_SFONT_0 XGL_CTX_SFONT_1 XGL_CTX_SFONT_2 XGL_CTX_SFONT_3
3D Context only	Transform (3D)	XGL_3D_CTX_NORMAL_TRANS
	Memory Raster	XGL_3D_CTX_SURF_BACK_FPAT
	Light	XGL_3D_CTX_LIGHTS
	Data Map Texture	XGL_3D_CTX_SURF_FRONT_DMAP XGL_3D_CTX_SURF_BACK_DMAP
	Texture Map	XGL_3D_CTX_SURF_FRONT_TMAP XGL_3D_CTX_SURF_BACK_TMAP

The following code fragment assumes that an application has already created a Raster Device object, two Stroke Font objects, and a Marker object and shows a Context object being associated with these objects using `xgl_object_set()`.

```
xgl_object_set(ctx,
               XGL_CTX_DEVICE,                ras,
               XGL_CTX_SFONT_0,               strokefont,
               XGL_CTX_SFONT_1,               strokefont1,
               XGL_CTX_MARKER,                appl_marker,
               XGL_CTX_MARKER_SCALE_FACTOR,   15.0,
               NULL);
```

Context Object Operators

This section presents information on the Context object operators that are not graphics drawing primitives. Table 6-2 summarizes these operators. For more information on these operators, see the *XGL Reference Manual*.

Table 6-2 Context Object Utility Operators

Operator	Description
`xgl_context_new_frame()`	Clears the Device Coordinate viewport associated with the specified Context (see Chapter 10,"View Model"). The operator can clear the Z-buffer (reset all Z values to a given value) when using a 3D Context, switch hardware display buffers in a multiple buffer configuration, or synchronize to the Device's vertical retrace.
`xgl_context_post()`	Causes any pending graphics primitives for the given Context to be posted. This operator is superseded by `xgl_context_flush()` and is available to provide backward compatibility.
`xgl_context_flush()`	Causes any pending graphics primitives for the given Context to be posted or any asynchronous processing to conclude. Depending on the value of the `flush_action` flag passed as a parameter with the call, this operator can flush the XGL system's use of an application's data or its internal buffers. It can also ensure synchronization of XGL and device processing by blocking XGL processing until the rendering device has posted its buffers.

Table 6-2 Context Object Utility Operators *(Continued)*

Operator	Description
`xgl_context_push()`	Saves the specified attributes from the current graphics state of the Context onto an internal stack. (Each Context has its own internal stack.) If the list of specified attributes is `NULL`, all pushable attributes associated with the Context are saved.
`xgl_context_pop()`	Restores the most recently pushed Context state from the internal stack of the given Context. All the attributes pushed by the most recent call to the `xgl_context_push()` operator are restored in the Context.
`xgl_context_set_pixel()`	Sets the color value of a pixel at a specified position in a buffer of the Raster associated with the given Context. The buffer is specified by the Context attribute `XGL_CTX_RENDER_BUFFER` and can be an image buffer or a Z-buffer.
`xgl_context_set_multi_pixel()`	Sets the color value of a list of pixels at specified positions in the specified buffer of the Raster associated with the given Context. The buffer is specified by the Context attribute `XGL_CTX_RENDER_BUFFER` and can be an image buffer or a Z-buffer.
`xgl_context_set_pixel_row()`	Sets the color value of a row of pixels in the specified buffer of the Raster associated with the given Context. The buffer is specified by the Context attribute `XGL_CTX_RENDER_BUFFER` and can be an image buffer or a Z-buffer.
`xgl_context_get_pixel()`	Gets the color value of a pixel at a specified position in a buffer of the Raster (Device) associated with the given Context. The buffer is specified by the Context attribute `XGL_CTX_RENDER_BUFFER`.
`xgl_context_copy_buffer()`	Copies a rectangular block of pixels from a buffer in the source Raster to one or more of the Raster buffers associated with the destination Context. The source buffer is specified by the attribute `XGL_RAS_SOURCE_BUFFER`. The destination buffer is specified by the Context attribute `XGL_CTX_RENDER_BUFFER`. Copying is only supported between buffers of the same type.
`xgl_image()`	Displays a block of pixels from an XGL Raster.
`xgl_context_update_model_trans()`	Used to perform fast push and pop operations involving the model transforms.

Context Stacks

There are two types of stacks for XGL Contexts. The first is a general stack that is manipulated with the `xgl_context_push()` and `xgl_context_pop()` operators. This type of stack is allocated in main memory, and there is generally no limit to the number of successive `xgl_context_push()` calls that an application program can make. Each `xgl_context_push()` can save the value of a single Context attribute or the entire Context state.

A list of the attributes this operator can save is located in the *XGL Reference Manual*, under the `xgl_context_push()` operator. When the application program invokes the `xgl_context_pop()` operator, the last set of values saved by the most recent call to `xgl_context_push()` is extracted from an internal stack into the Context state.

In some cases, faster push and pop operations are needed—for example, when traversing a list of structures, involving the modification of the model transforms of the Context, to animate a set of objects in an XGL Window Raster. In such cases, the `xgl_context_update_model_trans()` operator is best suited. It can push or pop the model transforms of the Context, using a different Context stack than the one used by the more general `xgl_context_push()` and `xgl_context_pop()` operators. This special stack must be initialized by the application before using `xgl_context_update_model_trans()`. The Context attribute `XGL_CTX_MODEL_TRANS_STACK_SIZE` serves that purpose. (See the *XGL Reference Manual* for more detail on how to use the `xgl_context_update_model_trans()` operator.)

Environment Context Attributes

This section provides an overview of the environment Context attributes. None of these attributes can be pushed by `xgl_context_push()`. For information on the graphics Context attributes, see “Graphics Context Attributes” on page 185.”

`XGL_CTX_DEFERRAL_MODE`

Defines the *deferral mode* of a Context. It controls whether the data sent to the Device are buffered (held temporarily) or rendered immediately. The deferral modes are:

- XGL_DEFER_ASTI (At Some TIme). In this mode, XGL keeps an internal buffer of primitives, rendering them all when the buffer is full. The application must call xgl_context_post to see the buffered primitives rendered on the Device.
- XGL_DEFER_ASAP (As Soon As Possible). In this mode, XGL renders a primitive as soon as it is received.

The default value is XGL_DEFER_ASTI.

XGL_CTX_DEVICE
Defines the Device associated with a given Context. The device can be a Window Raster or Memory Raster. It is the drawing surface for all following graphics operations for that Context. The default value is NULL.

XGL_CTX_RENDER_BUFFER
Controls where primitives are rendered on the Device associated with the Context. Possible values are:

- XGL_RENDER_DRAW_BUFFER. Primitive operations are rendered on the raster's draw buffer.
- XGL_RENDER_DISPLAY_BUFFER. Primitive operations are rendered on the raster's display buffer when the raster is a double-buffered window raster.
- XGL_RENDER_Z_BUFFER. This value is used only during raster operations and is only effective when the XGL_3D_CTX_HLHSR_MODE is set to XGL_HLHSR_Z_BUFFER.

The default value is XGL_RENDER_DRAW_BUFFER. Note that it is not advisable to use this attribute for operators other than xgl_context_copy_buffer().

XGL_CTX_RENDERING_ORDER
Allows a device to set the rendering order of primitives. Applications can use this attribute to batch primitives in the most efficient order for a device. If the rendering order is XGL_RENDERING_ORDER_GIVEN, no reordering of primitives occurs. If the rendering order is XGL_RENDERING_ORDER_HLHSR, the accelerator can change the rendering order only if XGL_3D_CTX_HLHSR_MODE is set to XGL_HLHSR_Z_BUFFER. If the rendering order is set to XGL_RENDERING_ORDER_ANY, the device can reorder the rendering of the primitives. The default value is XGL_RENDERING_ORDER_GIVEN.

XGL_CTX_GEOM_DATA_IS_VOLATILE

When set to FALSE, this attribute indicates that the application program guarantees that the geometry data will not be modified or destroyed until a subsequent xgl_context_flush() call. The default value for this attribute is TRUE.

Context Attributes for Picking

The following environment Context attributes specify how picking is handled. For more information on picking, see Chapter 14, "Picking".

XGL_CTX_PICK_BUFFER_SIZE

Sets the size of the Context pick buffer. The value is the number of *Xgl_pick_info* structures that the internal pick buffer will hold. The *Xgl_pick_info* structures hold picking identifiers that identify the picked primitives. The default value is 256.

XGL_CTX_PICK_ENABLE

Turns picking on (TRUE) or off (FALSE, the default value).

XGL_CTX_PICK_STYLE

Controls how the pick events (pick identifier pairs) are stored in the Context pick buffer. If the value of this attribute is XGL_PICK_FIRST_N, the first N (where N is the size of the pick buffer) pick events are stored in the buffer, all other pick events are not buffered. If the value of this attribute is XGL_PICK_LAST_N, the events are stored in a first-in-first-out (FIFO) manner. This attribute can only be changed when the attribute XGL_CTX_PICK_ENABLE is FALSE. The default value is XGL_PICK_LAST_N.

XGL_CTX_PICK_APERTURE

Defines the area (2D Context) or the volume (3D Context) specified in Device Coordinates, which is used to test primitives for picking. Primitives within the specified coordinates are picked when the mouse button is pushed. The default value for 2D is $[-1,1] \times [-1,1]$, and for 3D is $[-1,1] \times [-1,1] \times [-1,1]$.

XGL_CTX_PICK_SURF_STYLE

Controls how polygon picking occurs. If the value of this attribute is XGL_PICK_SURF_AS_SOLID, the polygon is picked as a solid polygon, regardless of the current setting of XGL_CTX_SURF_FRONT_FILL_STYLE or XGL_3D_CTX_SURF_BACK_FILL_STYLE. If this attribute is set to

XGL_PICK_SURF_AS_FILL_STYLE, the polygon is picked as it is rendered, in accordance with the XGL picking semantics. The default value is XGL_PICK_SURF_AS_SOLID.

Context Attributes Associated with the View Model

XGL_CTX_VDC_ORIENTATION

Defines the orientation of the Virtual Device Coordinate (VDC) space relative to Device Coordinate (DC) space. The coordinate system is always right-handed. (See Chapter 10, “View Model,” for more information.)

XGL_CTX_VDC_ORIENTATION enables the programmer to change the orientation of the VDC space to one of the following:

- XGL_Y_DOWN_Z_AWAY
- XGL_Y_UP_Z_TOWARD.

The default value is XGL_Y_DOWN_Z_AWAY. See Figure 10-3 on page 302 for more information.

XGL_CTX_MODEL_TRANS

Returns the Transform object that represents the Model Transform, which maps geometry from model coordinates to world coordinates.

XGL_CTX_MC_TO_DC_TRANS

Returns the Transform object that represents the Model-Coordinate-to-Device-Coordinate Transform, which maps geometry from model coordinates to device coordinates.

XGL_3D_CTX_NORMAL_TRANS

Returns the Transform object that represents the Normal Transform, which transforms normal vectors from model coordinates to world coordinates.

Context Attributes Associated with Lighting

XGL_3D_CTX_LIGHT_NUM

This attribute sets the number of Lights in a 3D Context. When an application sets this attribute, XGL changes the number of Lights and their corresponding on and off switches to the value of this attribute. The default value is 0. (See Chapter 15, “Lighting, Shading, and Depth Cueing” for more information.)

`XGL_3D_CTX_LIGHTS`

This attribute specifies an array of Lights used by a given Context. The default value is `NULL`.

Primitives and Graphics Context Attributes

7

This chapter discusses the XGL primitives and their associated Graphics Context attributes. It includes information on the following topics:

- XGL drawing primitive data structures
- Annotated primitives and text
- Raster-level primitives
- Accumulation buffers
- Examples demonstrating primitives

Introduction to XGL Drawing Primitives

Primitives are the basic graphics drawing routines that render geometry on a hardware device. XGL primitives render relatively simple geometric entities, such as lines, circles, arcs, and rectangles, and more complex entities, such as B-spline surfaces, quadrilateral mesh, and triangle strips. Geometry is rendered via the Context object; for this reason, the drawing primitives are considered to be Context operators. The dimension of the rendered geometry is determined by the Context object: 2D geometry requires 2D input data and is rendered through a 2D Context, and 3D geometry requires 3D input data and is rendered through a 3D Context. Most XGL primitives can be used with both 2D and 3D Context objects.

As in basic geometry, XGL deals with three kinds of geometric entities: points, lines, and surfaces. The primitives that XGL provides to draw these entities are summarized below and briefly described in Table 7-2 on page 173.

- Point primitive

  To render points and symbols, XGL provides the multimarker primitive. `xgl_multimarker()` takes a list of 2D or 3D points for the locations of the markers and then draws the specified marker symbol at those positions.

- Line primitives

  To render lines, XGL provides the multipolyline primitive. `xgl_multipolyline()` draws a list of unconnected 2D or 3D polylines, where each polyline is a series of line segments.

  To render curves, XGL provides a B-spline curve primitive. `xgl_nurbs_curve()` draws 2D or 3D non-uniform, rational or non-rational B-spline curves. The points of the B-spline point list are interpreted as the set of control points for the curve. Context attributes control the smoothness and other characteristics of the curve.

- Surface primitives

  A number of XGL primitives render surfaces (which are also known as filled areas, polygons, or closed regions). The surface perimeters defined by the geometry in the point list enclose an area (surface) that can be filled with color, patterns, or texture. Surface primitives can be rendered with shading or lighting, and the front and back faces of surfaces can be determined and rendered with different color, patterns, illumination characteristics, or texture. The following XGL primitives render surfaces:

Table 7-1 XGL Surface Primitives

Primitive	Description
`xgl_multirectangle()`	Renders a list of 2D or 3D rectangles.
`xgl_multicircle()`	Renders a list of 2D or 3D circles.
`xgl_multiarc()`	Renders a list of 2D or 3D arcs. 3D arcs are approximated as polygons.
`xgl_polygon()`	Renders a single-surface (planar) 2D or 3D polygon that can be defined by more than one boundary; a polygon can also be defined with holes.

Table 7-1 XGL Surface Primitives *(Continued)*

Primitive	Description
`xgl_multi_simple_polygon()`	Renders a list of 2D or 3D single-boundary polygons.
`xgl_multi_elliptical_arc()`	Renders a list of 3D elliptical arcs.
`xgl_triangle_strip()`	Renders a 3D triangle strip. Triangle strips, triangle lists, and quadrilateral meshes reduce the amount of geometry transferred to XGL by using individual points in more than one surface definition, if the surfaces are connected along their boundaries.
`xgl_triangle_list()`	Renders a 3D triangle list. Triangle lists can consist of connected triangles arranged as a triangle strip, connected triangles arranged as a triangle star, or unconnected triangles.
`xgl_quadrilateral_mesh()`	Renders a 3D quadrilateral mesh.
`xgl_nurbs_surface()`	Renders a NURBS surface as a set of triangles. The NURBS surface can be trimmed.

- Text primitives

  XGL provides two primitives to render stroke text and an associated primitive to determine the rectangle that encompasses a text string. The stroke text primitive, `xgl_stroke_text()`, renders 2D or 3D text as a set of polylines. The annotation text primitive, `xgl_annotation_text()`, also renders 2D or 3D text but generates the characters in a plane parallel to the display surface.

Note – Bitmap fonts can be rendered using the raster operators `xgl_context_copy_buffer()` and `xgl_image()`. For more information on raster text, see “Raster Text” on page 339.

Twelve of the available primitives are illustrated in Figure 7-1 on page 172.

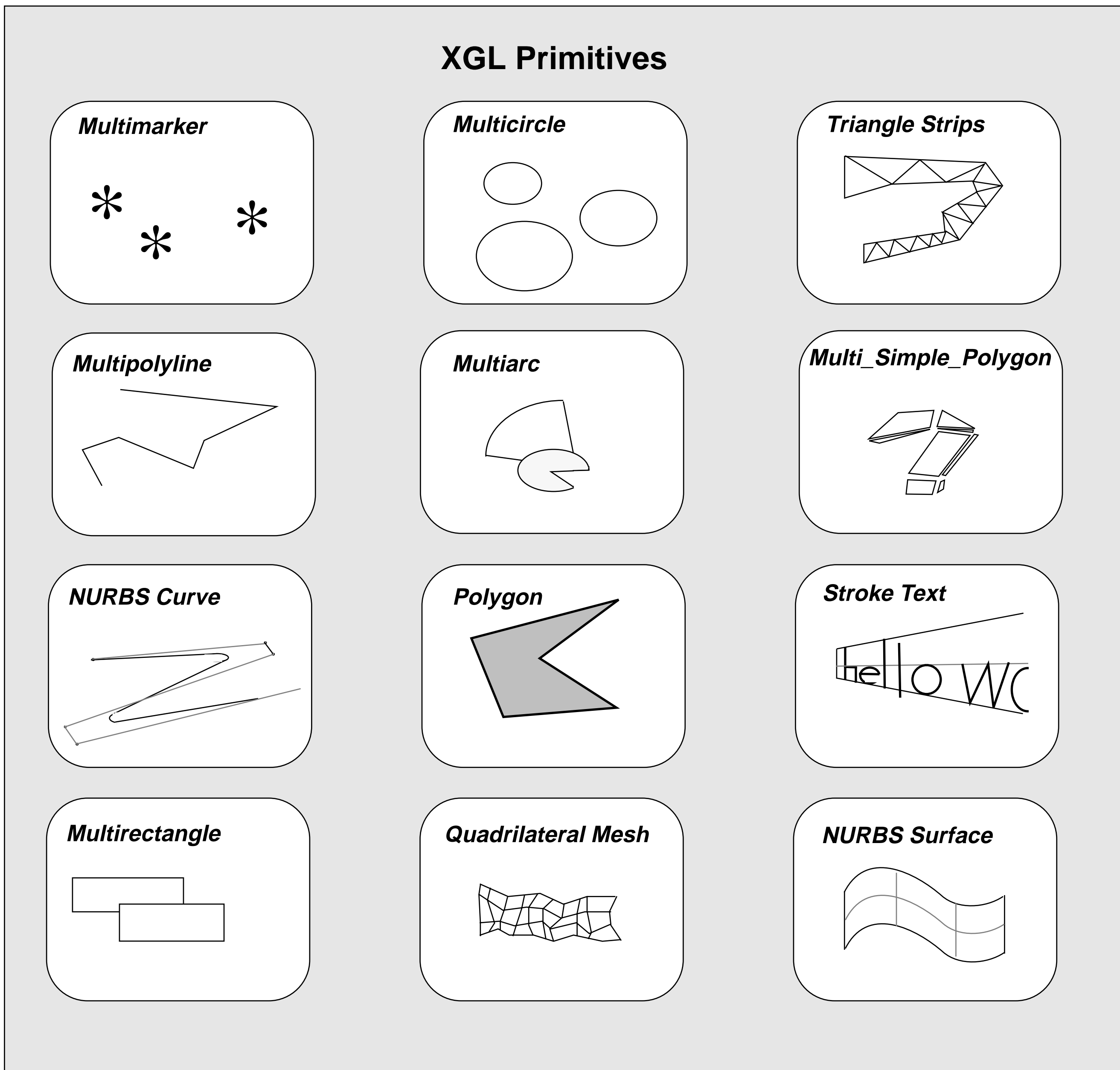


Figure 7-1 XGL Drawing Primitives Illustrated

Description of Drawing Primitives

Table 7-2 provides an overview of the input parameters required for each of the XGL geometric primitive operators. For more information on the input parameters for each operator, see the appropriate manual page in the *XGL Reference Manual.*

Table 7-2 Geometric Primitive Operators *(1 of 4)*

Operator	Description
void **xgl_annotation_text** (Xgl_ctx ctx, void *str, Xgl_pt_f*d *ref_pos, Xgl_pt_f*d *ann_pos);	Draws a string indexed by the characters in `str` using the Context `ctx`. Annotation text primitives differ from text primitives in that the plane upon which the characters are generated is always parallel to the display surface. Annotation text primitives are planar primitives. The Z-coordinate of the annotation point defines the depth of the annotation text.
void **xgl_multiarc** (Xgl_ctx ctx, Xgl_arc_list *arc_list);	Draws a list of arcs of a circle according to the graphics state described in the 2D or 3D Context. The arc traverses a counterclockwise (CCW) path from starting angle to stopping angle when the *x*-axis points right and the *y*- axis points up; this path is clockwise (CW) when the *x*-axis points right and the *y*-axis points down. The argument `arc_list` is a pointer to the list of arcs to draw. Each arc in the list is transformed by the current transformation matrix and drawn in the plane of the view surface.
void **xgl_multicircle** (Xgl_ctx ctx, Xgl_circle_list *circle_list);	Draws a list of circles according to the graphics state described in the Context `ctx`. The argument `circle_list` is a pointer to a list of circles to draw.
void **xgl_multi_elliptical_arc** (Xgl_ctx ctx, Xgl_ell_list *ell_list);	Draws a list of 3D elliptical arcs according to the graphics state described in the 3D Context. The `ell_list` argument is a pointer to the list of elliptical arcs to draw. Each elliptical arc in the list is transformed by the current transformation matrix and drawn in the plane described by the two direction vectors provided with each arc.

Table 7-2 Geometric Primitive Operators *(2 of 4)*

Operator		Description
void **xgl_multimarker** (Xgl_ctx Xgl_pt_list	 ctx, *pl);	Draws markers at a given list of 2D or 3D points. Characteristics of the markers are determined by the graphics state of the Context `ctx`. The argument `pl` is a pointer to the list of points where the markers are drawn.
void **xgl_multipolyline** (Xgl_ctx Xgl_bbox Xgl_usgn32 Xgl_pt_list	 ctx, *bbox, num_pt_lists, pl[]);	Draws a list of 2D or 3D unconnected polylines. Each point list describes one polyline. The argument `pl` is the first element in the array of point list structures that specify the vertices of the polylines.
void **xgl_multirectangle** (Xgl_ctx Xgl_rect_list	 ctx, *rect_list);	Draws a list of rectangles using the current surface attributes. The `rect_list` argument is a pointer to the list of rectangles to draw.
void **xgl_multi_simple_polygon**(Xgl_ctx Xgl_facet_flags Xgl_facet_list Xgl_bbox Xgl_usgn32 Xgl_pt_list	 ctx, flags, *facets, *bbox, num_pt_lists, pl[]);	Draws a set of simple polygons according to the parameter list and the graphics state described in a 2D or 3D Context. The `pl` argument is the first element of an array of point structures, each of which specifies the vertices of a simple polygon. The `facet` argument is a pointer to facet information about individual polygons.
void **xgl_nurbs_curve** (Xgl_ctx Xgl_nurbs_curve Xgl_bounds_f1d Xgl_curve_color_spline	 ctx, *curve, *range, *color_spline);	Draws a non-uniform, B-Spline (NURBS) curve. `curve` is a pointer to a structure defining the order of the curve, the number of knots, the knot vector, and a list of control points for the curve. The `range` argument is a pointer to a structure specifying which portion of the curve is actually displayed.

Table 7-2 Geometric Primitive Operators *(3 of 4)*

Operator		Description
void **xgl_nurbs_surface** (Xgl_ctx Xgl_nurbs_surf Xgl_trim_loop_list Xgl_nurbs_surf_simple_geom Xgl_surf_color_spline Xgl_surf_data_spline_list	 ctx, *surface, *trimming, *hints, *color_spline, *data_splines);	Draws a NURBS surface according to the parameter list and the graphics state described in a 3D Context. The surface is defined by a two-dimensional grid of control points, knot sequences in u,v parameter space, and trimming information specifying which portion of parameter space is actually rendered. Optional information can define the original geometry.
void **xgl_polygon** (Xgl_ctx Xgl_facet_type Xgl_facet Xgl_bbox Xgl_usgn32 Xgl_pt_list	 ctx, facet_type, *facet, *bbox, num_pt_lists, pl[]);	Draws a single polygon defined according to the parameter list and the graphics state defined in the Context `ctx`, which may be 2D or 3D. The `num_pt_lists` argument describes the number of boundaries in the multibounded polygon. The `pl` argument is the first element of an array of point lists. Each point list specifies the vertices of a single boundary of the polygon. `facet` is a pointer to facet information for individual polygons.
void **xgl_quadrilateral_mesh** (Xgl_ctx Xgl_usgn32 Xgl_facet_list Xgl_pt_list	 ctx, row_dim, col_dim, *facets, *pl);	Draws a quadrilateral (quad) mesh according to the parameter list and the graphics state described in a 3D Context. A quad mesh is defined by a single point list containing at least four vertices. The first four vertices define a quad; additional vertices define additional quads.
void **xgl_stroke_text** (Xgl_2d_ctx void Xgl_pt_f*d Xgl_pt_f3d	 ctx, *str, *pos, dir[]);	Draws a string indexed by the characters in `str` using the Context `ctx`. The `pos` argument is the 2D or 3D reference point for the rendered string. The `str` argument is a NULL-terminated, C-style string of characters. The `dir` argument is an array of two vectors that define the plane on which 2D text is located in 3D environments; this argument is ignored in 2D Contexts. Each point within the stroke character is transformed through the XGL transformation pipeline.

Table 7-2 Geometric Primitive Operators *(4 of 4)*

Operator		Description
void **xgl_stroke_text_extent** (Xgl_ctx void Xgl_bounds_f2d Xgl_pt_f2d	 ctx, *string, *rect, *cat_pt);	Computes a rectangle in text local coordinates bounding a stroke text string. The Context `ctx` contains the Stroke Font object and stroke text attributes necessary to compute the rectangular box returned in the argument `rect`. The `cat_pt` argument specifies where the string ends and can be used to append a new string to a previously rendered one.
void **xgl_triangle_list** (Xgl_ctx Xgl_facet_list Xgl_pt_list Xgl_tlist_flags	 ctx, *facets, *pl, flags);	Draws a triangle list according to the parameter list and the graphics state described in a 3D Context. A triangle list is defined by a single point list containing at least three vertices. The list may consist of a series of connected triangles arranged as a triangle strip, a series of connected triangles arranged as a triangle star, or a series of unconnected independent triangles. The first three vertices from the point list define a triangle. Flags passed to the primitive via the `flags` parameter or supplied in the flag word in the vertex data determine whether additional vertices produce a triangle strip, triangle star, independent triangles, or a combination of the three.
void **xgl_triangle_strip** (Xgl_ctx Xgl_facet_list Xgl_pt_list	 ctx, *facets, *pl);	Draws a triangle strip according to the parameter list and the graphics state described in a 3D Context. A triangle strip is defined by a single point list containing at least three vertices. The first three vertices define a triangle, and each additional vertex defines an adjacent triangle.

Drawing Primitive Data Structures

The geometry in XGL primitives is defined by a set of coordinates specified in the Local Modeling Coordinate system. XGL transforms the coordinates to the Device Coordinate system and draws the primitive at the given coordinates. Applications can supply coordinate point data in 2D integer or 2D or 3D floating point data types. Depending on the geometry to be drawn and on the Context, one point data type may be more appropriate than another.

XGL drawing primitives use a variety of data structures as operands in the function calls. The primary data structures are discussed in the following sections.

Point Types

Point data structures are used to supply drawing primitives with coordinate data. The point structures may also specify other information, such as color, flags that control the drawing of edges around surfaces, normals, or (*u,v*) data. The type of a point is defined by its ***dimension*** (2D, 3D, 2D homogeneous, or 3D homogeneous), its ***data type*** (int or float), and any additional information (a color, a normal, or a flag). See the *Xgl_pt_list* man page or the *xgl_struct* man page for information on the available point types in XGL.

Point Lists

Most XGL primitives take a list of points as an operand. Point lists store the coordinate values that define a primitive. When the coordinates in a point list are input to a primitive operator, they produce an image on the screen. The list of points is represented to XGL by the data structure *Xgl_pt_list*, which is defined as:

```
typedef struct {
    Xgl_pt_type         pt_type;
    Xgl_bbox_status     bbox_status;
    Xgl_usgn32          num_pts;
    Xgl_usgn32          num_data_values;
    Xgl_pt_pcnf         pts;
} Xgl_pt_list
```

The point list data structure has five fields: the type of points (all points in a list must have the same type), a bounding box (described in the next section), the number of points in the list, the number of data values at each point, and a pointer to the array of points.

Note – The primitives `xgl_multiarc()`, `xgl_multicircle()`, `xgl_multirectangle()`, and `xgl_multi_elliptical_arc()` take special data structures for drawing lists of arcs, circles, rectangles, ellipses, and elliptical arcs. These data structures are *Xgl_arc_list*, *Xgl_circle_list*, *Xgl_rect_list*, and *Xgl_ell_list*. They are similar to *Xgl_pt_list*. The primitive `xgl_multi_elliptical_arc()` is 3D only.

Facet Structures

Some surface primitives take an additional parameter: a *facet* structure, or a list of facet structures. Generally, a facet refers to a single planar section of a surface, which might be a simple polygon or one triangle within an XGL triangle strip. The facet data provide information such as a color or normal, or both, for that section of the surface. The 3D surface primitives that take facet structures are multi-simple polygons, general polygons, quadrilateral mesh, triangle strips, and triangle lists.

Facets are passed to XGL using the data structure *Xgl_facet_list*. The *Xgl_facet_list* data structure is:

```
typedef struct {
    Xgl_facet_type      facet_type;
    Xgl_usgn32          num_facets
    union {
        Xgl_color_facet          *color_facets;
        Xgl_normal_facet         *normal_facets;
        Xgl_color_normal_facet   *color_normal_facets;
    } facets;
} Xgl_facet_list
```

The facet list structure has three fields: the *type of facets* (all facets in a list must have the same type), the *number of facets* in the list, and a *pointer* to an array of facet data containing information about the facet color or normal vector. Facets are optional arguments; a facet structure can be set to `NULL`.

The color data in a facet structure specify how a facet is to be colored. When an application supplies more than one source of color, it can select the source of color (Context, facet, or vertex) to be used in rendering with the attribute XGL_CTX_SURF_FRONT_COLOR_SELECTOR. It can also select the illumination mode using the Context color, facet color, or vertex color. Color facet data can be supplied for a 2D or a 3D surface.

The normal vectors provided with facet data allow XGL to determine whether a facet on a 3D surface is front-facing or back-facing. Information on the orientation of a facet is needed for lighting, face culling, or face distinguishing. A facet normal points in the direction of the front face of the surface. Facet normals provided to XGL should be unit vectors.

An application can supply normal vectors per facet in the facet data structures, or it can supply normals per vertex in the point data structures. When facet data is not explicitly provided for surface primitives as part of an application's data, the XGL system calculates the facet normal for a surface using the Context attribute XGL_3D_CTX_SURF_GEOM_NORMAL. This attribute computes the geometric facet normal by selecting three vertices on the surface and determining whether the vertices are ordered in a clockwise or counterclockwise manner. For more information on how XGL calculates facet normals, see the man page for XGL_3D_CTX_SURF_GEOM_NORMAL.

Surface primitives that do not take facet structures as input require information about vectors as part of the point structures input to the primitive. These primitives are 3D multicircles, multiarcs, multirectangles, and multielliptical arcs. Fields in the 3D point structures for these primitives require orthogonal direction vectors to define the plane of the geometry. A normal vector for the surface can also be specified, or XGL will compute a normal vector to the surface using the vectors that define the plane. For more information on how the geometric normal is computed for a particular primitive, see the man page for the primitive.

Bounding Boxes

A bounding box is the smallest box that completely encloses all the geometry in a point list. A bounding box can be used to optimize clipping operations when rendering a primitive. If the entire bounding box is within the clip bounds, the image needs no clipping.

The bounding box data structures contain information about the minimum and maximum limits of the bounding box. The application has several ways to pass bounding box information to the XGL primitives. The application can:

- Pass bounding box information using the *Xgl_bbox* structure. This structure was available at the previous release of the XGL library and is retained to provide backward compatibility.
- Pass bounding box information using any of six new bounding box structures. These new structures reduce allocated memory and maintain compatibility with the previous release.
- Determine the status of the bounding box geometry and pass the status to the XGL primitive.

Each of these approaches is discussed in more detail in the sections below.

A bounding box is optional for all drawing primitives taking a point list. In addition, primitives taking a list of point lists (to render multiple geometric entities such as polylines) use an additional bounding box parameter, which specifies the bounding box of all the point list bounding boxes. The bounding box and the geometry are defined in the same coordinate system. The application can set the bounding box field to NULL if it does not need this data.

Providing Bounding Box Information Using the Xgl_bbox Structure

In previous releases of the XGL library, bounding box information was provided via a pointer to a structure of type *Xgl_bbox*. The *Xgl_bbox* structure is defined as:

```
typedef struct {
    Xgl_bbox_type      bbox_type;
    union {
        Xgl_bounds_i2d i2d;
        Xgl_bounds_f2d f2d;
        Xgl_bounds_f3d f3d;
    } box;
} Xgl_bbox;
```

For compatibility, the XGL library supports the *Xgl_bbox* structure, although this is no longer the recommended way to pass bounding box information to XGL primitives.

Providing Bounding Box Information Using the New XGL Bounding Box Structures

The current release of the XGL library provides six new structures in which to store bounding box information. These structures are similar to *Xgl_bbox*, but they enable the application to tailor memory usage to its specific requirements. The six structures are:

Xgl_bbox_i2d	*Xgl_bbox_status*
Xgl_bbox_f2d	*Xgl_bbox_d2d*
Xgl_bbox_f3d	*Xgl_bbox_d3d*

Each structure allocates only the memory needed. For example, *Xgl_bbox_f2d* is defined as:

```
typedef struct {
    Xgl_bbox_type      bbox_type;
    union {
        Xgl_bounds_f2d f2d;
    } box;
} Xgl_bbox_f2d;
```

In place of an *Xgl_bbox* structure, the application should use one of the new structures. However, if the `bbox` field in a point list points to a structure of a type other than *Xgl_bbox*, the application must cast the variable to *Xgl_bbox*

before compiling. For example, if an application uses a variable called `bbox_f3d` of type ***Xgl_bbox_f3d***, it must cast the pointer to ***Xgl_bbox*** before passing the information to XGL, as shown in the code fragment below.

```
{
    Xgl_pt_list         pl;
    Xgl_bbox_f3d        bbox_f3d;

    pl.bbox = (Xgl_bbox *) (&bbox_f3d);
}
```

Providing Bounding Box Geometry Status

The application can optimize clipping operations by determining whether the geometry will be clipped before calling the drawing primitive. If the application wants to pass the geometry status to an XGL primitive, it can first call `xgl_context_check_bbox()` to determine the geometry status of the bounding box and then pass the status to the primitive using the ***Xgl_bbox_status*** structure. The `xgl_context_check_bbox()` operator enables the application to calculate the geometry status of the bounding box. This operator determines whether the geometry that is to be input to a drawing primitive will be completely clipped, partially clipped, or not clipped (inside the drawing area) after it is transformed to device coordinates. The primitive is defined as:

```
void xgl_context_check_bbox (
    Xgl_ctx                         ctx,
    Xgl_primitive_type              prim_type,
    Xgl_bbox                        *bbox,
    Xgl_geom_status                 *geom_status);
```

For polygonal primitives, if the application knows that the primitives contained in the bounding box are all front facing (or back facing), it can provide this information to the XGL primitive by logically OR'ing the flag `XGL_GEOM_STATUS_FACING_FRONT` (or `XGL_GEOM_STATUS_FACING_BACK`) with the geometry status calculated by `xgl_context_check_bbox()`, before passing the geometry status to XGL drawing operators.

The following code fragment is an example of how to use `xgl_context_check_bbox()` to calculate the geometry status of a bounding box and pass the status to an XGL primitive.

```
{
  Xgl_ctx             ctx;
  Xgl_primitive_type prim_type;
  Xgl_bbox_f2d        bbox_f2d;
  Xgl_bbox_status     bbox_status;
  Xgl_geom_status     geom_status;
  Xgl_pt_list         pl;

  prim_type = XGL_PRIM_MULTIMARKER;

  /* set up the bounding box for the 2d markers */
  bbox_f2d.bbox_type = XGL_BBOX_F2D;
  bbox_f2d.box.f2d.xmin = bbox_f2d.box.f2d.ymin = 100.0;
  bbox_f2d.box.f2d.xmax = bbox_f2d.box.f2d.ymax = 200.0;

 xgl_context_check_bbox(ctx, prim_type, (Xgl_bbox *)(&bbox_f2d),
                             &geom_status);

  /* set up the Xgl_bbox_status structure */
  bbox_status.bbox_type = XGL_BBOX_STATUS;
  bbox_status.box.status = geom_status;

  pl.bbox = (Xgl_bbox *)(&bbox_status);
  xgl_multimarker(ctx, &pl);
  }
```

The application must ensure that the bounding box's geometry status is valid for the geometry. Thus, if any of the XGL Context attributes affecting the Model Coordinate to Device Coordinate transformation and clipping (or affecting the front/back facing status) is modified, the bounding box geometry status must be recalculated.

Note – The geometry status calculated by `xgl_context_check_bbox()` only applies to XGL non-Gcache primitives such as `xgl_multimarker()`. For XGL Gcache operators, such as `xgl_gcache_multimarker()`, applications should use one of the other bounding box structures.

Annotated Primitives

An application has the option of rendering certain primitives in a plane parallel to the projection plane. Primitives rendered in this way are called annotation primitives. Annotation primitives are 3D multicircles, arcs, elliptical arcs, multirectangles, and annotation text. The multicircle, arc, elliptical arc, and multirectangle operators generate annotated primitives when the point list data structures input to the operators specify annotation data.

`xgl_multicircle()`
: To generate annotation multicircles, specify type `XGL_MULTICIRCLE_AF3D` for the data type of the circle's defining points and use a point type of *Xgl_circle_af3d* to define the circle point structures. The center of each circle represents the reference point and is specified in model coordinates. The radius is in virtual device coordinates. There is no need for direction vectors, since the primitive is always front-facing.

`xgl_multiarc()`
: To generate annotation arcs, specify type `XGL_MULTIARC_AF3D` for the data type of the arc's defining points and use a point type of *Xgl_arc_af3d* to define the arc point structures. The center of each arc represents the reference point and is specified in model coordinates. The radius is in virtual device coordinates. There is no need for direction vectors, since the primitive is always front-facing.

`xgl_multi_elliptical_arc()`
: To generate annotation elliptical arcs, specify type `XGL_MULTIELLIARC_AF3D` for the data type of the arc's defining points and use a point type of *Xgl_ell_af3d* to define the elliptical arc point structures. The center of each elliptical arc represents the reference point and is specified in model coordinates. The major and minor axes are specified in virtual device coordinates. There is no need for direction vectors, since the primitive is always front-facing.

`xgl_multirectangle()`
: To generate annotation rectangles, specify type `XGL_MULTIRECT_AF3D` for the data type of the rectangle's defining points and use a point type of *Xgl_rect_af3d* to define the rectangle point structures. The reference corner of each rectangle is specified in model coordinates, and the other corner represents the virtual device coordinates. There is no need for direction vectors, since the primitive is always front-facing.

The annotation text primitive always generates text parallel to the projection plane. See Chapter 11, “Text”, for information on annotation text.

Graphics Context Attributes

Rendering describes how graphics data is drawn — how polygons are filled and lines are styled, as well as how clipping and transformations are performed. Rendering is defined by the attributes set within the Context object. Some attributes (like the `XGL_CTX_ROP` [Raster OPeration] mode) apply to all primitives, while others apply to specific primitives.

Sometimes the value of a Context attribute can be overridden during a call to a primitive operator. For example, the points in a polyline can have color values that are used for each line segment. These point color values can override the polyline color value. In addition, object attributes may interact with Context attributes to produce the final rendered appearance of geometry. For example, the Line Pattern object determines the pattern of the rendered lines or edges, while the Context line attributes determine the appearance of the lines or edges. As another example, even though the color of surfaces is set with a Context surface attribute, the Light object can modify the surface color of a primitive.

The Context attributes that contribute to the display of primitives are listed and briefly described in the sections below. The relationship between rendering primitives and Context attributes during the execution of a primitive is illustrated in Figure 7-2 on page 202. The sections below are intended merely as an overview of the function that the Context object encompasses. For information on specific Context attributes, see the subsequent chapters of this manual or the *XGL Reference Manual*.

General Rendering Attributes

The Context attributes in Table 7-3 apply to most or all of the drawing primitives.

Table 7-3 General Context Rendering Attributes

Attribute	Description
XGL_CTX_DEFERRAL_MODE	Determines whether graphics data sent to a Device is buffered. Primitives can be rendered as soon as possible (`XGL_DEFER_ASAP`) or rendered at a later time (`XGL_DEFER_ASTI`). In the latter case, primitives are rendered either when the buffer is full or when the operator `xgl_context_flush()` is called.
XGL_CTX_THRESHOLD	Defines a threshold value for a primitive's size, below which the primitive is simplified before rendering.
XGL_CTX_PLANE_MASK	Defines the pixel-plane mask.
XGL_CTX_ROP	Defines how the set of pixels comprising the graphics primitive is logically combined with the destination Raster's pixels to product the output written to the Raster Device.
XGL_CTX_BACKGROUND_COLOR	Defines the color used for background pixels.
XGL_CTX_RASTER_FILL_STYLE	Defines the fill style used for the Raster-level primitive `xgl_context_copy_buffer`.
XGL_CTX_RASTER_FPAT	Defines the Raster fill pattern used when `xgl_context_copy_buffer` is invoked using a stipple fill.
XGL_CTX_RASTER_FPAT_POSITION	Defines the mapping of the fill pattern when `xgl_context_copy_buffer` is invoked using a stipple fill.
XGL_CTX_RASTER_STIPPLE_COLOR	Defines the background color used when `xgl_context_copy_buffer` is invoked using an opaque stipple fill.
XGL_CTX_NEW_FRAME_ACTION	Sets the actions performed by the `xgl_context_new_frame` operator.

Table 7-3 General Context Rendering Attributes *(Continued)*

Attribute	Description
XGL_CTX_RENDER_BUFFER	Controls where primitives are rendered on the Device associated with the Context. Primitive operations can be rendered onto the Raster draw buffer, the image buffer, or the Z buffer.
XGL_CTX_RENDERING_ORDER	Allows a device to set the rendering order of primitives.
XGL_CTX_GEOM_DATA_IS_VOLATILE	Specifies whether the application's data can be used after the XGL primitive returns.

Context Accumulation Buffer Attributes

Context attributes specify the destination buffer for an accumulation operation and specify the amount of jittering for the accumulation. When jittered images are averaged together, the resulting image is effectively antialiased. Table 7-4 lists the Context accumulation buffer attributes. See page 222 for information on accumulation buffers.

Table 7-4 Context Accumulation Buffer Attributes

Attributes	Description
XGL_3D_CTX_ACCUM_OP_DEST	Specifies the buffer to be used as the destination buffer during an accumulation operation.
XGL_3D_CTX_JITTER_OFFSET	Specifies the amount by which geometry should be offset in device coordinates before drawing. This is used when rendering an image before accumulating into an accumulation buffer.

Hidden Line/Hidden Surface Removal (HLHSR)

In a 3D Context, the application programmer can use XGL_3D_CTX_HLHSR_MODE to enable the XGL hidden line and surface removal features. Z-buffering is used for all hidden line and surface removal. If the output device doesn't support hardware Z-buffering, a Z-buffer is created in system memory. However, simulating a hardware Z-buffer in this way is slower than using direct hardware support.

Hidden line and surface removal is defined with the Context attributes listed in Table 7-5.

Table 7-5 Context Hidden Line and Surface Removal Attributes

Attribute	Description
XGL_3D_CTX_HLHSR_MODE	Defines the mode used for hidden line and hidden surface removal.
XGL_3D_CTX_HLHSR_DATA	Defines the data used to clear the Z-buffer for hidden line/hidden surface mode.
XGL_3D_CTX_Z_BUFFER_WRITE_MASK	Defines which bit planes of the Z-buffer are written.
XGL_3D_CTX_Z_BUFFER_COMP_METHOD	Specifies the comparison technique used when the Z-buffer is updated. The result of the comparison determines how the destination buffer is updated.

Technique for Fast Hidden Line Removal

If only hidden line removal is needed, and the output device doesn't support hardware Z-buffering, the application programmer can use the attribute XGL_CTX_SURF_EDGE_FLAG to produce a fast, hidden line-removal technique. To use this technique, follow these general steps:

1. Set the XGL_CTX_SURF_EDGE_FLAG attribute to ON.
2. Set the front and back surface colors to the background color. (See the man pages for XGL_3D_CTX_SURF_BACK_COLOR, XGL_3D_CTX_SURF_FRONT_COLOR, and XGL_3D_CTX_BACKGROUND_COLOR.)

3. Set shading and depth cueing to XGL_ILLUM_NONE and XGL_DEPTH_CUE_OFF, respectively. (See the man pages for XGL_3D_CTX_SURF_BACK_ILLUMINATION, XGL_3D_CTX_SURF_FRONT_ILLUMINATION, and XGL_3D_CTX_DEPTH_CUE_MODE.)

The rendering is equivalent to a fast, hidden line removal technique, preserving correct readability of the displayed data.

Context Line Attributes

Context line attributes, listed in Table 7-6, set line style and line color. They also define the shape of the endpoints of lines and curves, set the line width scale factor, and define the shape of joins between line segments. Line primitives are polylines and B-spline curves. For information on line patterns, see Chapter 12, "Line Patterns."

Table 7-6 Context Line Attributes

Attribute	Description
XGL_CTX_LINE_ALT_COLOR	Defines the alternate color used in patterned lines.
XGL_CTX_LINE_AA_BLEND_EQ XGL_CTX_LINE_AA_FILTER_SHAPE XGL_CTX_LINE_AA_FILTER_WIDTH	Control whether lines are antialiased and defines the method of antialiasing.
XGL_CTX_LINE_CAP	Defines the shape of the endpoints of lines and curves.
XGL_CTX_LINE_COLOR	Defines the color of lines and curves.
XGL_CTX_LINE_COLOR_SELECTOR	Determines whether the source of a line's color is the Context or the primitive vertex data.
XGL_CTX_LINE_JOIN	Defines the shape of joins between line segments.
XGL_CTX_LINE_MITER_LIMIT	Defines the limit for mitering of line segment corners.
XGL_CTX_LINE_PATTERN	Defines the pattern used for patterned lines.
XGL_CTX_LINE_STYLE	Defines the style of a line, that is, whether the line is solid or patterned.

Table 7-6 Context Line Attributes *(Continued)*

Attribute	Description
XGL_CTX_LINE_WIDTH_SCALE_FACTOR	Defines the line width scale factor used to determine the width of lines and curves.
XGL_3D_CTX_LINE_COLOR_INTERP	Controls color interpolation along 3D lines.

Note – Solid and dashed wide lines are not displayed when viewed on end, but a point is displayed.

Context Marker Attributes

Context attributes define the size, color, and type of marker that is rendered. Context Marker rendering attributes are listed in Table 7-7. For information on Marker objects, see Chapter 13, “Markers.”

Table 7-7 Context Marker Attributes

Attribute	Description
XGL_CTX_MARKER	Defines the Marker description drawn by `xgl_multimarker`.
XGL_CTX_MARKER_AA_BLEND_EQ XGL_CTX_MARKER_AA_FILTER_SHAPE XGL_CTX_MARKER_AA_FILTER_WIDTH	Control whether markers are antialiased and defines the method of antialiasing.
XGL_CTX_MARKER_COLOR	Defines the color of markers.
XGL_CTX_MARKER_COLOR_SELECTOR	Determines whether the source of a marker’s color is the Context or the geometry point data.
XGL_CTX_MARKER_SCALE_FACTOR	Defines the Marker scale factor used to determine the width of Markers.

Context Curve Attributes

The attributes in Table 7-8 define NURBS curve rendering. Context line attributes also apply to NURBS curves. For more information on NURBS curves, see Chapter 8, “Rendering NURBS Curves and Surfaces with XGL.”

Table 7-8 Context NURBS Curve and Surface Attributes

Attribute	Description
XGL_CTX_NURBS_CURVE_APPROX	Defines the NURBS curve approximation method.
XGL_CTX_NURBS_CURVE_APPROX_VAL	Defines the NURBS curve approximation values.
XGL_CTX_MIN_TESSELLATION XGL_CTX_MAX_TESSELLATION	Defines the minimum and maximum number of segments in which a NURBS or 3D conic curve can be tessellated when using the primitives `xgl_nurbs_curve`, `xgl_multicircle`, `xgl_multiarc`, and `xgl_multi_elliptical_arc`.

Context Surface Attributes

Context attributes define many of the aspects of surface rendering. For example, Context attributes set the color used to fill surfaces and define the way in which surfaces are filled. Context attributes specify how a surface is shaded and illuminated. They also define how surface normals are calculated when the normals are not provided in application data.

Surface primitives are rectangles, circles, circular and elliptical arcs, polygons, quadrilateral mesh, triangle strips, triangle lists, and NURBS surfaces. Some surface rendering attributes can be specified for both 2D and 3D Contexts, while other attributes are only used in 3D Contexts or in 2D Contexts. A brief overview of the surface rendering attributes is provided in the sections below. The 3D surface characteristics related to lighting and shading are discussed more fully in Chapter 15, “Lighting, Shading, and Depth Cueing.”

Surface Edge Rendering Attributes

Edges are characteristics of the surfaces of primitives; they enclose a surface and are drawn if the surface edge attribute XGL_CTX_SURF_EDGE_FLAG is set to TRUE. Some primitives have additional control over edge rendering: if the edge flag attribute is set to enable edge rendering, primitives that accept point data that includes edge flag information can use the edge flag information to control the rendering of edge boundaries on a per-vertex basis.

If edge rendering is enabled, Context attributes set the edge style, color, and width characteristics of the edges of surfaces. Edge rendering attributes are listed in Table 7-9.

Table 7-9 Context Edge Attributes

Attribute	Description
XGL_CTX_SURF_EDGE_FLAG	Controls whether edges are drawn around surfaces.
XGL_CTX_EDGE_AA_BLEND_EQ XGL_CTX_EDGE_AA_FILTER_SHAPE XGL_CTX_EDGE_AA_FILTER_WIDTH	Control whether surface edges are antialiased and defines the method of antialiasing.
XGL_CTX_EDGE_ALT_COLOR	Defines the alternate color used in patterned edges.
XGL_CTX_EDGE_CAP	Defines the shape of teh endpoints of an edge.
XGL_CTX_EDGE_COLOR	Defines the color used for edges of surfaces.
XGL_CTX_EDGE_JOIN	Defines the shape of the joins between edge segments.
XGL_CTX_EDGE_MITER_LIMIT	Defines the limit for mitering of edge segment corners.
XGL_CTX_EDGE_PATTERN	Specifies the line pattern used for patterned edges.
XGL_CTX_EDGE_STYLE	Defines the edge style used for edges of surfaces.
XGL_CTX_EDGE_WIDTH_SCALE_FACTOR	Defines the edge width scale factor used to determine the width of surface edges.
XGL_3D_CTX_EDGE_Z_OFFSET	Specifies the offset added to a surface to define the DC location of the surface edge.

Surface Color, Light, and Fill Attributes

Context attributes control the global rendering properties of surfaces, such as material color, and the reflection coefficients for the various components of reflected light. The front and back faces of 3D geometry can be determined and shaded or lit appropriately. If 3D surface face distinguishing is enabled by setting the attribute `XGL_3D_CTX_SURF_FACE_DISTINGUISH` to `TRUE`, the front-facing surfaces are rendered using the `XGL_3D_CTX_SURF_FRONT_` attributes, and the back-facing surfaces are rendered using the `XGL_3D_CTX_SURF_BACK_` attributes. Otherwise, both front and back surfaces are rendered with the `XGL_3D_CTX_SURF_FRONT_` or `XGL_CTX_SURF_FRONT_` attributes. 2D surfaces are always rendered as front surfaces. Table 7-10 provides an overview of the Context surface rendering attributes.

Table 7-10 Context Surface Attributes

Attribute	Description
XGL_CTX_ARC_FILL_STYLE	Defines how arcs are filled.
XGL_CTX_SURF_AA_BLEND_EQ XGL_CTX_SURF_AA_FILTER_SHAPE XGL_CTX_SURF_AA_FILTER_WIDTH	Controls whether surfaces are antialiased and defines the method of antialiasing. These attributes only control the appearance of hollow antialiased surfaces. To antialiase filled surfaces, see “Accumulation Buffer” on page page 222.
XGL_3D_CTX_SURF_FACE_DISTINGUISH	Controls whether front- and back-facing surfaces are distinguished from each other.
XGL_CTX_FRONT_COLOR XGL_3D_CTX_SURF_BACK_COLOR	Defines the color used to fill surfaces.
XGL_CTX_SURF_FRONT_COLOR_SELECTOR XGL_3D_CTX_SURF_BACK_COLOR_SELECTOR	Selects the source of a surface’s color from the Context or from facet or vertex data provided with the primitive.
XGL_CTX_SURF_FRONT_FILL_STYLE XGL_3D_CTX_SURF_BACK_FILL_STYLE	Defines how surfaces are filled.
XGL_CTX_SURF_FRONT_FPAT XGL_3D_CTX_SURF_BACK_FPAT	Defines the fill pattern raster used for stipple fill.
XGL_CTX_SURF_FRONT_FPAT_POSITION XGL_3D_CTX_SURF_BACK_FPAT_POSITION	Defines the origin of the fill pattern raster used for stipple fill.

Table 7-10 Context Surface Attributes *(Continued)*

Attribute	Description
XGL_3D_CTX_SURF_FRONT_AMBIENT XGL_3D_CTX_SURF_BACK_AMBIENT	Defines the ambient reflection coefficients used for lighting calculations within a 3D Context.
XGL_3D_CTX_SURF_FRONT_DIFFUSE XGL_3D_CTX_SURF_BACK_DIFFUSE	Defines the diffuse reflection coefficients used for lighting calculations within a 3D Context.
XGL_3D_CTX_SURF_FRONT_ILLUMINATION XGL_3D_CTX_SURF_BACK_ILLUMINATION	Specifies how illumination calculations are performed.
XGL_3D_CTX_SURF_FRONT_LIGHT_ COMPONENT XGL_3D_CTX_SURF_BACK_LIGHT_COMPONENT	Determines which components of the lighting equation are used to compute a surface color. Any combination of the components ambient, diffuse, or specular can be enabled.
XGL_3D_CTX_SURF_FRONT_SPECULAR XGL_3D_CTX_SURF_BACK_SPECULAR	Defines the specular coefficient used in computing lighting.
XGL_3D_CTX_SURF_FRONT_SPECULAR_COLOR XGL_3D_CTX_SURF_BACK_SPECULAR_COLOR	Defines the specular color used in computing lighting.
XGL_3D_CTX_SURF_FRONT_SPECULAR_POWER XGL_3D_CTX_SURF_BACK_SPECULAR_POWER	Defines the specular power used in computing lighting.
XGL_3D_CTX_SURF_FACE_CULL	Controls face culling in a 3D Context.
XGL_3D_CTX_SURF_GEOM_NORMAL	Controls how surface normals are calculated for a facet if the surface normals are not provided as part of the application data.
XGL_3D_CTX_SURF_LIGHTING_NORMAL_FLIP	Defines how surface normals are treated for lighting.
XGL_3D_CTX_SURF_NORMAL_FLIP	Specifies whether vertex and facet normals are flipped 180 degrees.
XGL_3D_CTX_LIGHT_SWITCHES	Defines the on and off switches that correspond to the 3D Context's array of lights. The number of switches is given by `XGL_3D_CTX_LIGHT_NUM`.
XGL_CTX_SURF_INTERIOR_RULE	Defines the way in which pixels are determined to be inside or outside polygons with multiple bounds.

Table 7-10 Context Surface Attributes *(Continued)*

Attribute	**Description**
XGL_3D_CTX_SURF_DC_OFFSET	Determines whether a DC offset is added to the z component of each vertex of a polygon. The offset value moves the polygon back from its computed depth position.
XGL_3D_CTX_SURF_SILHOUETTE_EDGE_FLAG	Controls whether silhouette edges are drawn around a surface.

Surface Transparency Attributes

Context attributes enable applications to render transparent surfaces. XGL provides screen-door and blended (two-pass) transparency. If the transparency method is screen-door, a device-dependent *screen door* (a mesh that allows rendering of only some of the surface's pixels) is applied to the primitive. If the transparency method is blended, blending equations determine how the surface pixel values are blended with the background color or existing pixel values. Transparent surfaces are subject to hidden line and surface removal as defined by `XGL_3D_CTX_HLHSR_MODE`. Transparency is supported for 3D surface primitives only. Table 7-11 lists the surface transparency attributes.

Table 7-11 Context Surface Transparency Attributes

Attribute	**Description**
XGL_3D_CTX_SURF_TRANSP_METHOD	Defines the method used to render transparent surfaces. Methods are `XGL_TRANSP_BLENDED` or `XGL_TRANSP_SCREEN_DOOR`.
XGL_3D_CTX_SURF_FRONT_TRANSP XGL_3D_CTX_SURF_BACK_TRANSP	Specifies the transparency value for surface primitives. Surface transparency can range from fully transparent through translucent to opaque.
XGL_3D_CTX_SURF_TRANSP_BLEND_EQ	Controls how the pixel values in transparent surfaces are blended with existing pixel values when the surface transparency method is `XGL_TRANSP_BLENDED`.
XGL_3D_CTX_BLEND_DRAW_MODE	Controls the drawing of blended and non-blended primitives.
XGL_3D_CTX_BLEND_FREEZE_Z_BUFFER	Defines whether blended primitives update the Z-buffer.

Curved Surface Attributes

Attributes for NURBS surface rendering are listed in Table 7-12. Context surface attributes also apply to NURBS surfaces. For information on NURBS surfaces, see Chapter 8, “Rendering NURBS Curves and Surfaces with XGL.”

Table 7-12 Context Curved Surface Attributes

Attribute	Description
XGL_CTX_NURBS_SURF_APPROX	Defines the NURBS surface approximation method.
XGL_CTX_NURBS_SURF_APPROX_VAL_U XGL_CTX_NURBS_SURF_APPROX_VAL_V	Defines the NURBS surface approximation values.
XGL_CTX_NURBS_SURF_PARAM_STYLE	Defines the appearance of the NURBS surface.
XGL_CTX_NURBS_SURF_ISO_CURVE_ PLACEMENT	Defines the placement of isoparametric curves on a NURBS surface.
XGL_CTX_NURBS_SURF_ISO_CURVE_U_NUM XGL_CTX_NURBS_SURF_ISO_CURVE_V_NUM	Defines the number of isoparametric curves in the u or v directions on a NURBS surface.
XGL_CTX_MIN_TESSELLATION XGL_CTX_MAX_TESSELLATION	Defines the minimum and maximum number of segments in which a NURBS surface can be tessellated.

Surface Texturing Attributes

Attributes for applying textures to 3D surfaces are listed in Table 7-13. For information on texture mapping and texture mapping attributes, see Chapter 17, “Texture Mapping.”.

Table 7-13 Context Texture Mapping Attributes

Attribute	Description
XGL_3D_CTX_SURF_FRONT_TMAP XGL_3D_CTX_SURF_BACK_TMAP	Specifies a list of Texture Map objects for a 3D Context.
XGL_3D_CTX_SURF_FRONT_TMAP_NUM XGL_3D_CTX_SURF_BACK_TMAP_NUM	Specifies the number of Texture Map objects in a 3D Context.
XGL_3D_CTX_SURF_FRONT_TMAP_SWITCHES XGL_3D_CTX_SURF_BACK_TMAP_SWITCHES	Specifies the list of on-off switches for a 3D Context’s Texture Map objects.

Table 7-13 Context Texture Mapping Attributes

Attribute	Description
XGL_3D_SURF_TMAP_PERSP_CORRECTION	Specifies the method used to compute texture coordinate values for surface interiors.
XGL_3D_CTX_SURF_FRONT_DMAP XGL_3D_CTX_SURF_BACK_DMAP	Specifies a list of Data Map Texture objects for a 3D Context.
XGL_3D_CTX_SURF_FRONT_DMAP_NUM XGL_3D_CTX_SURF_BACK_DMAP_NUM	Specifies the number of Data Map Texture objects in a 3D Context.
XGL_3D_CTX_SURF_FRONT_DMAP_SWITCHES XGL_3D_CTX_SURF_BACK_DMAP_SWITCHES	Specifies the list of on-off switches for a 3D Context's Data Map Texture objects.

Surface Depth Cueing Attributes

Context attributes define the type of depth cueing rendered, specify the color that the primitive will be modulated to as the depth increases, and define how the depth cueing interpolation is calculated. See Chapter 15, "Lighting, Shading, and Depth Cueing" for information on depth cueing. Depth cueing attributes are listed in Table 7-14.

Table 7-14 Context Surface Depth Cueing Attributes

Attribute	Description
XGL_3D_CTX_DEPTH_CUE_COLOR	For RGB Rasters, sets the Context's depth cue color, which is blended with the original colors of primitives to give the modulated colors of primitives after depth cueing.
XGL_3D_CTX_DEPTH_CUE_INTERP	Specifies whether interpolation according to calculated values of depth-cued color is to be performed.
XGL_3D_CTX_DEPTH_CUE_MODE	Defines the depth cueing mode in the Context. This can be set to: `XGL_DEPTH_CUE_OFF`, `XGL_DEPTH_CUE_LINEAR`, or `XGL_DEPTH_CUE_SCALED`.
XGL_3D_CTX_DEPTH_CUE_REF_PLANES	Specifies an array of two *z* values representing the front and the back planes. Used only for scaled depth cueing.
XGL_3D_CTX_DEPTH_CUE_SCALE_FACTORS	Scaling factors at front and back reference planes. Used only for scaled depth cueing.

Context Stroke Text Attributes

Context text attributes define stroke text character height, width, spacing, and alignment. They also define stroke text horizontal and vertical alignment, direction, and color. Table 7-15 lists stroke text attributes. For more information on the Stroke Text object, see Chapter 11, “Text.”

Table 7-15 Context Stroke Text Attributes

Attribute	Description
XGL_CTX_STEXT_AA_BLEND_EQ XGL_CTX_STEXT_AA_FILTER_SHAPE XGL_CTX_STEXT_AA_FILTER_WIDTH	Control whether stroke text is antialiased and defines the method of antialiasing.
XGL_CTX_SFONT_0 , XGL_CTX_SFONT_1, XGL_CTX_SFONT_2, XGL_CTX_SFONT_3	Specifies the Stroke Font for the current Context.
XGL_CTX_STEXT_CHAR_ENCODING	Specifies the character encoding scheme to be used with stroke text.
XGL_CTX_STEXT_CHAR_EXPANSION_FACTOR	Defines the horizontal expansion applied to stroke font text.
XGL_CTX_STEXT_CHAR_HEIGHT	Defines the nominal size of stroke text characters.
XGL_CTX_STEXT_CHAR_SLANT_ANGLE	Defines the angle that stroke text makes with the vertical direction.
XGL_CTX_STEXT_CHAR_SPACING	Defines the size of the gap between characters.
XGL_CTX_STEXT_CHAR_UP_VECTOR	Defines the up vector for stroke text.
XGL_CTX_STEXT_ALIGN_HORIZ	Defines the horizontal alignment for stroke text.
XGL_CTX_STEXT_ALIGN_VERT	Defines the vertical alignment for stroke text.
XGL_CTX_STEXT_COLOR	Defines the color used for rendering stroke text.
XGL_CTX_STEXT_PATH	Defines the direction in which stroke text is rendered.
XGL_CTX_STEXT_PRECISION	Defines the precision used for rendering stroke text.

Context Annotation Text Attributes

Annotation text is parallel to the display surface. Context attributes define annotation text character height, character alignment, text alignment, and text direction. Table 7-16 lists annotation text attributes. For more information on annotation text, see Chapter 11, “Text.”

Table 7-16 Context Annotation Text Attributes

Attribute	Description
XGL_CTX_ATEXT_CHAR_HEIGHT	Defines the nominal size of annotation text.
XGL_CTX_ATEXT_CHAR_SLANT_ANGLE	Defines the angle that annotation text with respect to the vertical direction.
XGL_CTX_ATEXT_CHAR_UP_VECTOR	Defines the up vector for annotation text.
XGL_CTX_ATEXT_STYLE	Defines the style in which annotation text is rendered.
XGL_CTX_ATEXT_ALIGN_HORIZ	Defines the horizontal alignment for annotation text.
XGL_CTX_ATEXT_ALIGN_VERT	Defines the vertical alignment for annotation text.
XGL_CTX_ATEXT_PATH	Defines the direction in which annotation text is rendered.

Context View and Transform Attributes

Transform objects are manipulated via Context object attributes. Context Transform attributes direct the conversion of geometric data from application coordinates to device coordinates. Table 7-17 lists Transform and viewing attributes. For more information on the Transform object, see Chapter 9, “Transforms.” For information on the XGL view model, see Chapter 10, “View Model.”

Table 7-17 Context View and Transform Attributes

Attribute	Description
XGL_CTX_VDC_ORIENTATION	Defines the orientation of virtual device coordinate (VDC) space.
XGL_CTX_MODEL_TRANS	Returns a read-only copy of the XGL Model Transform.
XGL_CTX_LOCAL_MODEL_TRANS	Defines the Transform object used to transform geometry data from local model coordinates to global modeling coordinates.
XGL_CTX_GLOBAL_MODEL_TRANS	Defines the Transform object used to transform geometry data from global model coordinates to world coordinates.
XGL_CTX_VIEW_TRANS	Defines the Transform object used to transform geometry data from world coordinates to virtual device (VDC) coordinates.
XGL_CTX_VIEW_CLIP_BOUNDS	Defines the VDC clipping area.
XGL_CTX_VDC_WINDOW	Defines the VDC window.
XGL_CTX_DC_VIEWPORT	Defines the DC viewport in which rendering occurs.
XGL_CTX_CLIP_PLANES	Defines which planes are used in view clipping.
XGL_CTX_VDC_MAP	Controls mapping of geometry data from virtual device coordinates to device coordinates.
XGL_CTX_MC_TO_DC_TRANS	Returns a read-only copy of the Transform object used to transform data from global model coordinates to device coordinates.
XGL_3D_CTX_NORMAL_TRANS	Returns a read-only copy of the Normal Transform, which transforms normal vectors from model coordinates to world coordinates.

Table 7-17 Context View and Transform Attributes *(Continued)*

Attribute	Description
XGL_3D_CTX_MODEL_CLIP_PLANE_NUM	Specifies the number of model clip planes in use.
XGL_3D_CTX_MODEL_CLIP_PLANES	Specifies the list of model clip planes to use.
XGL_3D_CTX_VIEW_CLIP_PLUS_W_ONLY	Controls clipping in homogeneous space.
XGL_CTX_MODEL_TRANS_STACK_SIZE	Defines the depth of the Transform stack associated with the Context. Used with the `xgl_context_update_model_trans` operator.

Context Paint Attributes

Context paint attributes specify the paint characteristics of a transparent overlay window. Table 7-18 lists the paint attributes.

Table 7-18 Context Paint Type Attributes

Attribute	Description
XGL_CTX_PAINT_TYPE	Specifies the paint type for a transparent overlay window.
XGL_CTX_NEW_FRAME_PAINT_TYPE	Specifies the paint type to be applied to an overlay window when the overlay window is cleared by `xgl_context_new_frame`.

Figure 7-2 on page 202 shows the rendering pipeline for the Context attributes. The figure illustrates the relationship between operators and attributes, and shows the order in which attributes affect the rendering of the final image.

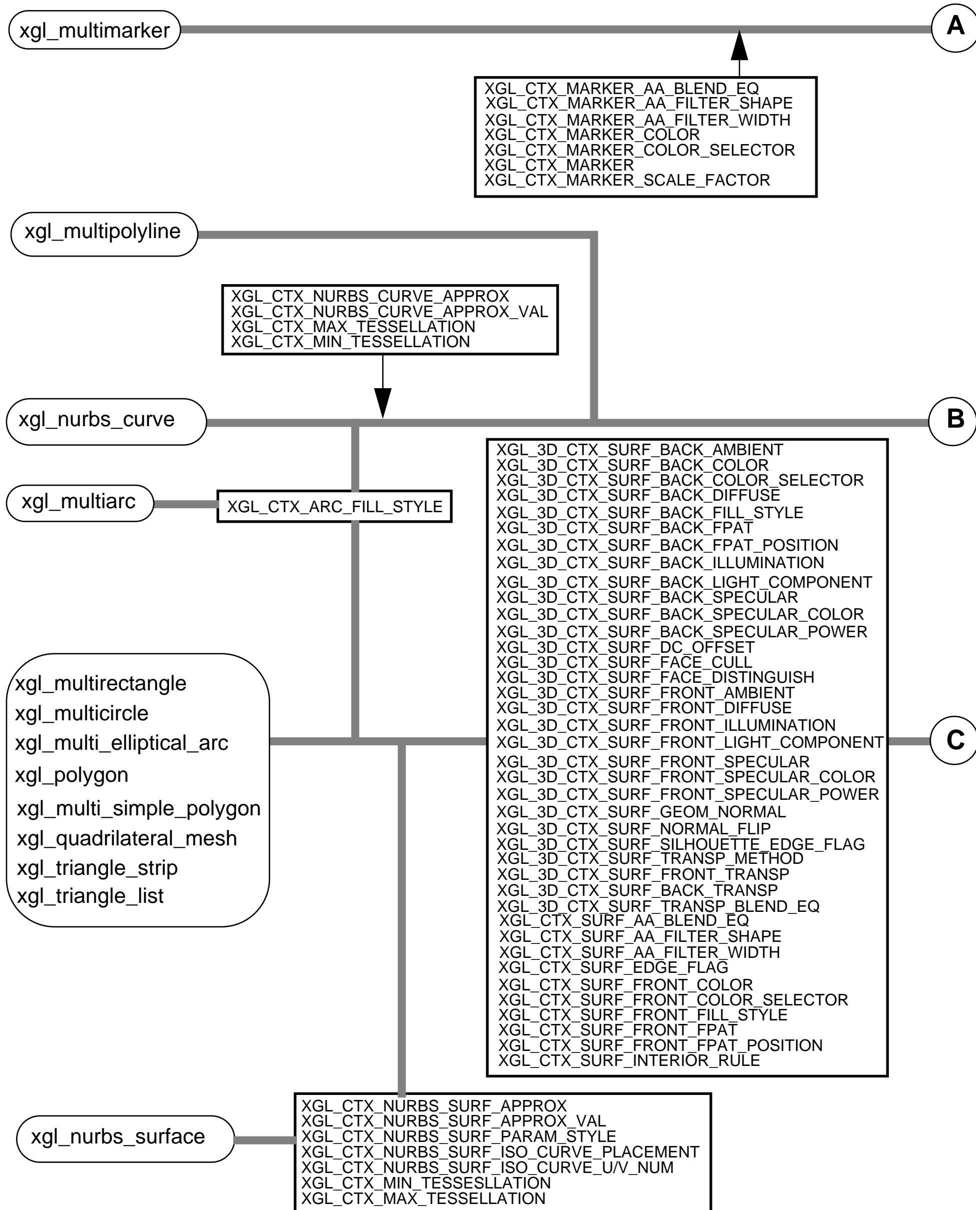


Figure 7-2 Context Attributes Pipeline

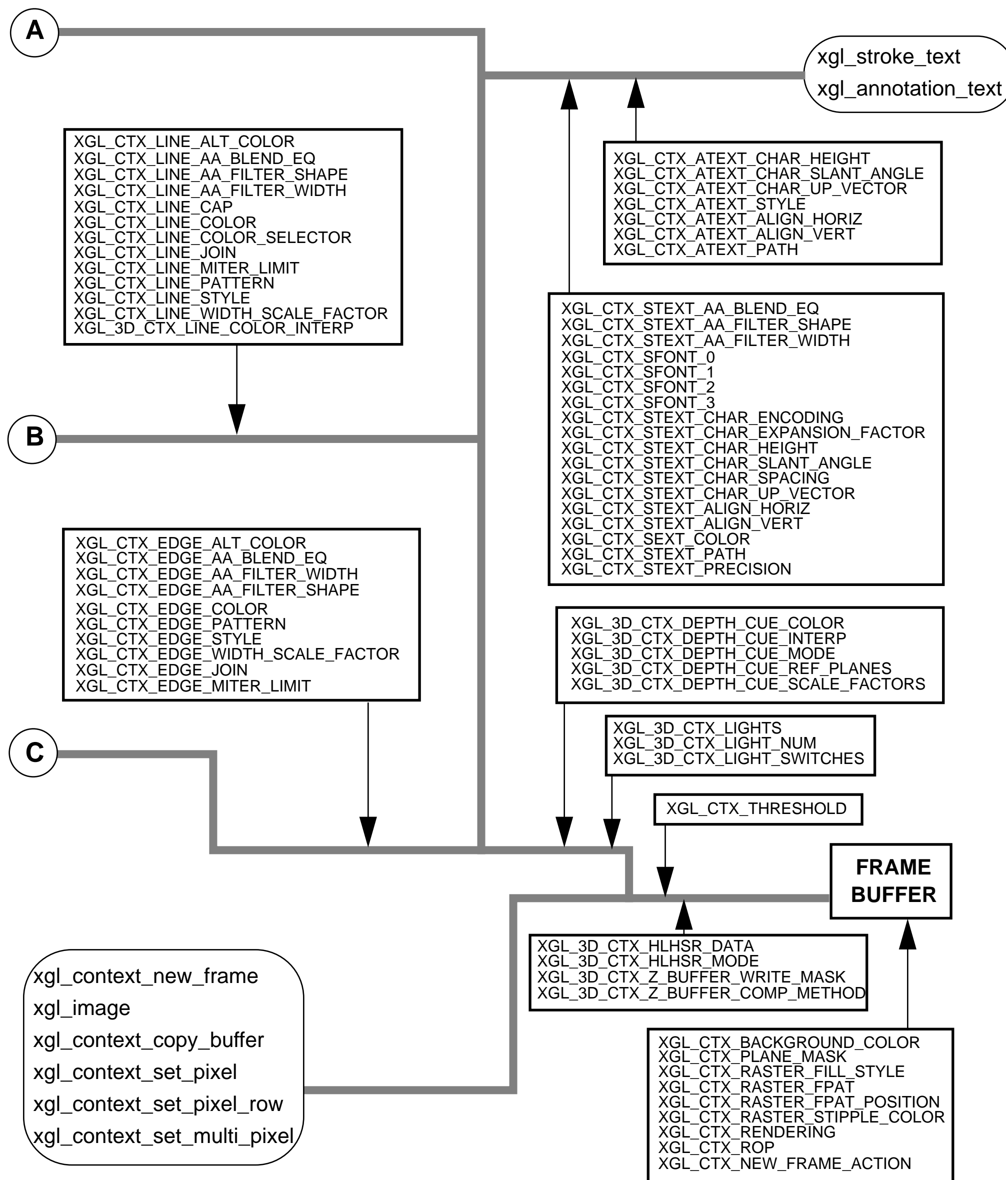


Figure 7-2 Context Attributes Pipeline (Continued)

Example Programs using XGL Primitives

The following example programs illustrate XGL primitives. Each example is a fragment of a larger program. The complete program includes `ex_utils.c` and `prims_2d_main.c`, both of which are listed in Appendix B, as well as all the examples listed in this section. To compile the complete program, type `make prims_2d` in the example program directory. The compiled program allows you to look at all the primitive example programs.

Polygon Example Program

This program, `prims_2d_pgon.c`, illustrates the XGL polygon primitive. Figure 7-3 shows the program output.

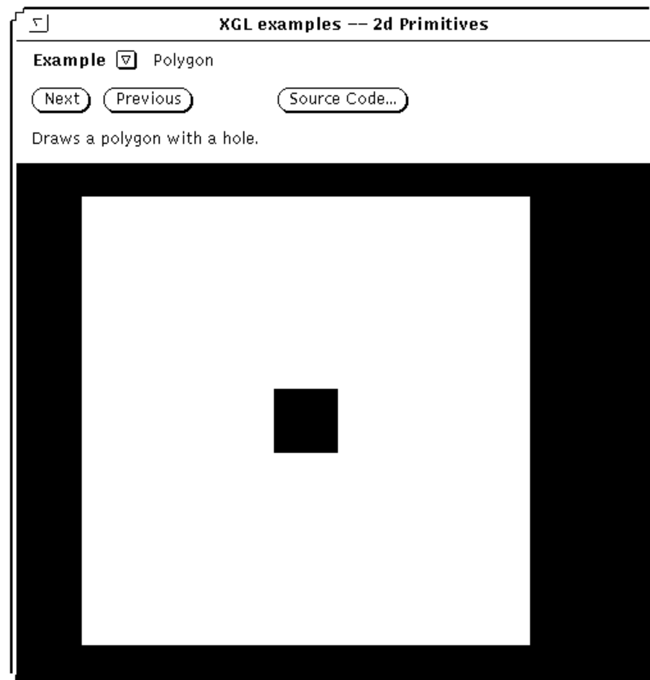


Figure 7-3 Output of `prims_2d_pgon.c`

Code Example 7-1 Polygon Example

```
/*
 * prims_2d_pgon.c
 */

#include "ex.h"

prims_2d_pgon (Xgl_object  ctx)
{
    Xgl_pt_i2d            pts1[20];
    Xgl_pt_i2d            pts2[20];
    Xgl_pt_list           pl[2];
    Xgl_color             color;

    /* points defining outer boundary of polygon */

    pts1[0].x =  50;
    pts1[0].y =  25;
    pts1[1].x = 400;
    pts1[1].y =  25;
    pts1[2].x = 400;
    pts1[2].y = 375;
    pts1[3].x =  50;
    pts1[3].y = 375;

    pl[0].pt_type = XGL_PT_I2D;
    pl[0].bbox = NULL;
    pl[0].num_pts = 4;
    pl[0].pts.i2d = pts1;

    /* points defining inner boundary of polygon */

    pts2[0].x = 200;
    pts2[0].y = 175;
    pts2[1].x = 250;
    pts2[1].y = 175;
    pts2[2].x = 250;
    pts2[2].y = 225;
    pts2[3].x = 200;
    pts2[3].y = 225;

    pl[1].pt_type = XGL_PT_I2D;
    pl[1].bbox = NULL;
    pl[1].num_pts = 4;
    pl[1].pts.i2d = pts2;
```

```
    /* draw cyan, two-boundary polygon */

    color = cyan_color;
    xgl_object_set (ctx,
           XGL_CTX_SURF_FRONT_COLOR, &color,
           NULL);

    xgl_polygon (ctx, XGL_FACET_NONE, NULL, NULL, 2, pl);
}
```

Rectangle Example Program

This program illustrates the XGL rectangle primitive. The program results are shown in Figure 7-4.

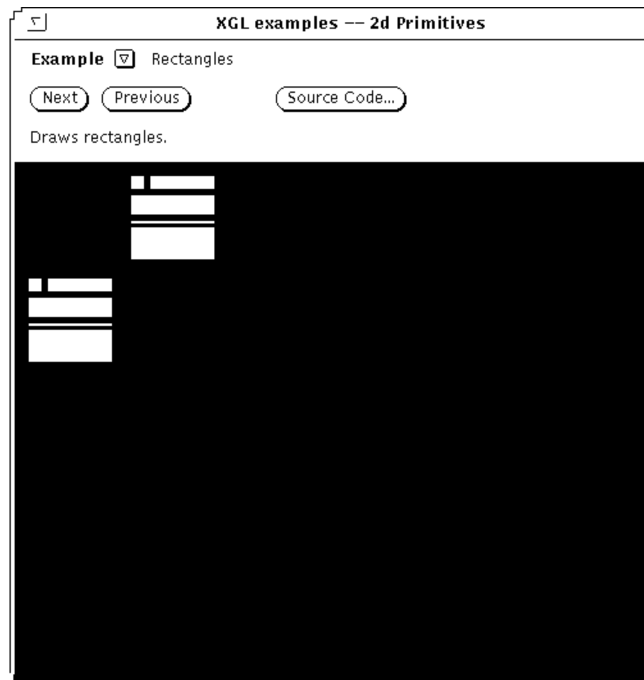


Figure 7-4 Output of `prims_2d_rect.c`

Code Example 7-2 Multirectangle Example

```
/*
 * prims_2d_rect.c
 */
#include "ex.h"

/* the 5 rectangles that will be drawn as a group several times */
static  Xgl_rect_i2d  rects[5] = {
                {{ 10, 10, 0 }, { 20, 20 }},
                {{ 25, 10, 0 }, { 75, 20 }},
                {{ 10, 25, 0 }, { 75, 40 }},
                {{ 10, 45, 0 }, { 75, 47 }},
                {{ 10, 50, 0 }, { 75, 75 }}
                          };

prims_2d_rect (Xgl_objectctx;
{
    Xgl_rect_list     rectlist;
    Xgl_color         color;
    Xgl_pt            pt;
    Xgl_pt_f2d        pt_f2d;
    Xgl_object        lmt;

    /* initialize rectlist struct */

    rectlist.rect_type = XGL_MULTIRECT_I2D;
    rectlist.num_rects = 5;
    rectlist.bbox = 0;
    rectlist.rects.i2d = rects;

    /* initialize pt struct used for translation */

    pt.pt_type = XGL_PT_F2D;
    pt.pt.f2d = &pt_f2d;

    /* draw 5 rectangles 1st time...all blue */

    color = blue_color;
    xgl_object_set (ctx, XGL_CTX_SURF_FRONT_COLOR, &color, NULL);

    xgl_multirectangle (ctx, &rectlist);

    /*
     * Get local model transform so that we can translate the
     * group of rectangles
     */
```

```
    xgl_object_get (ctx, XGL_CTX_LOCAL_MODEL_TRANS, &lmt);

   /* translate rectangle group 80 pixels to right */

    pt_f2d.x = 80.0;
    pt_f2d.y = 0.0;
    xgl_transform_translate (lmt, &pt, XGL_TRANS_REPLACE);

    /* draw 5 rectangles 2nd time...all yellow */

    color = yellow_color;
    xgl_object_set (ctx, XGL_CTX_SURF_FRONT_COLOR, &color, 0);

    xgl_multirectangle (ctx, &rectlist);

    /* translate rectangle group 80 pixels down */

    pt_f2d.x = 0.0;
    pt_f2d.y = 80.0;
    xgl_transform_translate (lmt, &pt, XGL_TRANS_REPLACE);

    /* draw 5 rectangles 3rd time...all cyan */

    color = cyan_color;
    xgl_object_set (ctx, XGL_CTX_SURF_FRONT_COLOR, &color, 0);
    xgl_multirectangle (ctx, &rectlist);

    /* translate rectangle group 250 pixels to right and down */

    pt_f2d.x = 250.0;
    pt_f2d.y = 250.0;
    xgl_transform_translate (lmt, &pt, XGL_TRANS_REPLACE);

    /* draw 5 rectangles 4th time...all red */

    color = red_color;
    xgl_object_set (ctx, XGL_CTX_SURF_FRONT_COLOR, &color, 0);

    xgl_multirectangle (ctx, &rectlist);
    /* restore local modeling transform */
    xgl_transform_identity (lmt);
}
```

Circle Example Program

This program illustrates the circle primitive. Figure 7-5 shows the program output.

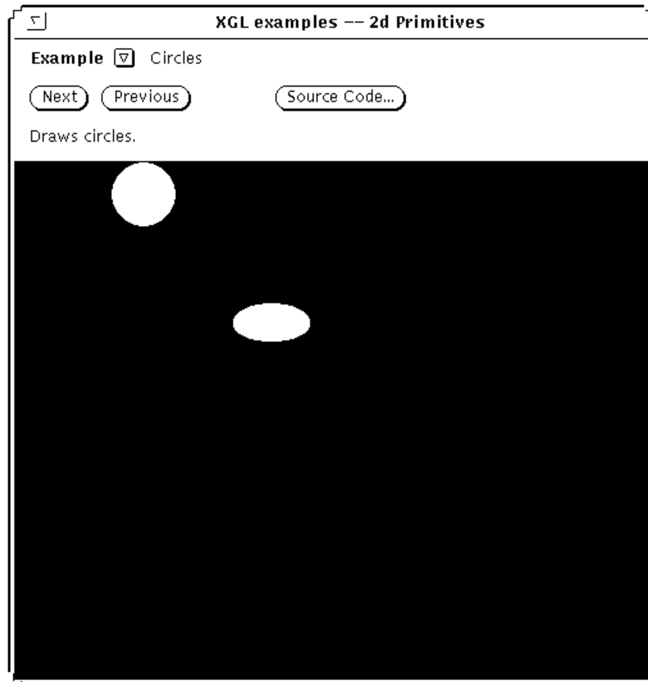


Figure 7-5 Output of `prims_2d_circle.c`

Code Example 7-3 Multicircle Example

```
/*
 * prims_2d_circle.c
 */

#include <math.h>
#include "ex.h"

Xgl_circle_f2d          circs[5] = {
                                {{ 25., 25., 0 },  25.},
                                {{ 25., 25., 0 },  15.},
                                {{ 25., 25., 0 },  5.},
                                {{ 100., 25., 0 }, 25.},
                                {{ 100., 25., 0 }, 15.}
                        };

extern render_ellipse(Xgl_object, Xgl_pt_f2d*, Xgl_pt_f2d*,
                      Xgl_pt_f2d*);


prims_2d_circle (Xgl_object  ctx)
{
    int                   i;
    Xgl_color             color, colors[5];
    Xgl_circle_list       circle_list;
    Xgl_pt_f2d            center, p1, p2;

    colors[0] = red_color;
    colors[1] = green_color;
    colors[2] = blue_color;
    colors[3] = magenta_color;
    colors[4] = yellow_color;

    /*
     * get a smooth circle
     */
    xgl_object_set(ctx, XGL_CTX_CURVE_APPROX, XGL_CURVE_METRIC_VDC,
                   XGL_CTX_CURVE_APPROX_VALUE, 1.0,
                   NULL);

    /* draw the circles */
    for (i = 0; i < 5; i++) {
        circle_list.type = XGL_MULTICIRCLE_F2D;
        circle_list.num_circles = 1;
        circle_list.bbox = 0;
```

```
        circle_list.circles.f2d = &circs[i];

        color = colors[i];
        xgl_object_set (ctx, XGL_CTX_SURF_FRONT_COLOR, &color,
                     NULL);

        xgl_multicircle(ctx, &circle_list);
    }

    /* draw the ellipses */
    for (i = 0; i < 5; i++) {
        center.x = circs[i].center.x + 100.;
        center.y = circs[i].center.y + 100.;
        p1.x = center.x;
        p1.y = center.y + ((float)(5 - i) * 5. + 20.);
        p2.x = center.x;
        p2.y = center.y + ((float)(5 - i) * 5. + 5.);

        color = colors[i];
        xgl_object_set (ctx, XGL_CTX_SURF_FRONT_COLOR, &color,
                     NULL);

        render_ellipse(ctx, &center, &p1, &p2);
    }
}

/*
 * modify the Xgl 2D transformations to render an ellipse by
 * using the circle primitive
 * center is the center point of the ellipse
 * p1 is the major axis and p2 is the minor axis
 */
render_ellipse(
    Xgl_object  ctx,
    Xgl_pt_f2d  *center,
    Xgl_pt_f2d  *p1,
    Xgl_pt_f2d  *p2)
{
    Xgl_circle_list     circle_list;
    Xgl_circle_f2d      f2d;
    Xgl_object          lmt;
    float               maj_axis, min_axis, frad;
    Xgl_pt              pt;
    Xgl_pt_f2d          ptf2d;

#define SQR(x)  ((x) * (x))
```

```
    maj_axis = (float)sqrt(SQR(p1->x - center->x) +
                         SQR(p1->y - center->y));
    min_axis = (float)sqrt(SQR(p2->x - center->x) +
                         SQR(p2->y - center->y));
    frad = maj_axis;

    xgl_object_get (ctx, XGL_CTX_LOCAL_MODEL_TRANS, &lmt);

    pt.pt_type = XGL_PT_F2D;
    ptf2d.x = -center->x;
    ptf2d.y = -center->y;
    pt.pt.f2d = &ptf2d;
    xgl_transform_translate(lmt, &pt, XGL_TRANS_REPLACE);
    pt.pt_type = XGL_PT_F2D;
    ptf2d.x = maj_axis / frad;
    ptf2d.y = min_axis / frad;
    pt.pt.f2d = &ptf2d;
    xgl_transform_scale(lmt, &pt, XGL_TRANS_POSTCONCAT);

    pt.pt_type = XGL_PT_F2D;
    ptf2d.x = center->x;
    ptf2d.y = center->y;
    pt.pt.f2d = &ptf2d;
    xgl_transform_translate(lmt, &pt, XGL_TRANS_POSTCONCAT);

    f2d.center.x = center->x;
    f2d.center.y = center->y;
    f2d.center.flag = 0;
    f2d.radius = frad;

    circle_list.num_circles = 1;
    circle_list.type = XGL_MULTICIRCLE_F2D;
    circle_list.bbox = 0;
    circle_list.circles.f2d = &f2d;
    xgl_multicircle(ctx, &circle_list);

    xgl_transform_identity(lmt);
}
```

Raster Primitives

Primitives are available from within the XGL Context object for performing low-level operations on the contents of the associated raster. These primitives are used to manipulate raster contents on a per-pixel basis. They enable the application to interact directly with the pixel values stored in the raster. These primitives were introduced in Chapter 6, “Contexts.”

Getting and Setting Pixel Values

The primitives `xgl_context_set_pixel()` and `xgl_context_get_pixel()` are used to set and retrieve the color value of a single pixel stored in a specified buffer of an XGL raster. Arguments to these calls are the Context associated with the Raster, the location of the pixel, and the color value. The pixel color value is assumed to be the same color type as the Raster color type (`XGL_DEV_COLOR_TYPE`). The buffer is specified by the Context attribute `XGL_CTX_RENDER_BUFFER`.

The primitive `xgl_context_set_multi_pixel()` is comparable in functionality to `xgl_context_set_pixel()`, but it accepts an array of pixels instead of just one pixel. `xgl_context_set_pixel_row()` sets multiple pixels of the same row to a color.

Copying Buffer Contents

The operator `xgl_context_copy_buffer()` copies a rectangular block of pixels from a buffer in the source raster to one or more of the Raster buffers associated with the destination Context. In the simple case of copying from a Memory Raster with one buffer to a non-multibuffered Window Raster, the pixels are copied from the draw buffer of the source Raster to the draw buffer of the Raster associated with the destination Context.

For more complicated cases with multiple buffers, the application program must set attributes to specify the buffer to be used as the source buffer and the buffer to be used as the destination in the Raster. The buffer to be used as source in the source Raster is specified by the source raster’s attribute `XGL_RAS_SOURCE_BUFFER`. The buffer(s) to be used as destination in the Raster associated to the destination Context is determined by the destination context attribute `XGL_CTX_RENDER_BUFFER`. The entire source raster or a portion of it can be copied to the destination raster. The area that is copied is specified by a

rectangular region in the source buffer. It is copied into the destination buffer starting at the specified position. If the position is a NULL pointer, the top and left corner position is used. If the rectangle is a NULL pointer, the maximum area from the source buffer is copied.

Copying is only supported between buffers of the same type; in other words, pixels can only be copied between image buffers or between Z-buffers. When copying between image buffers, the destination Context plane mask, fill rule, and raster operation mode (XGL_CTX_ROP) attributes are applied during the copy operation. When copying between Z buffers, the destination Context Z comparison method (specified by the attribute XGL_3D_CTX_Z_BUFFER_COMP_METHOD) and Z write mask (specified by the attribute XGL_3D_CTX_Z_BUFFER_WRITE_MASK) are applied during the copy operation.

Note – When copying to and from a Memory Raster, be sure that the Memory Raster has a Color Map that corresponds to its color type (XGL_DEV_COLOR_TYPE).

Note also that if the copy is from an 8-bit indexed Memory Raster to a 24-bit indexed Window Raster, separate Color Map objects are must be associated with the Rasters so that the XGL system can perform internal color map conversion correctly.

The operator xgl_image() copies a rectangular block of pixels from a source Memory Raster to the raster associated with the destination Context, which can be either a Memory Raster or a Window Raster. The operator performs the same operation as xgl_context_copy_buffer() does except that the location the pixels are copied to in the destination raster is expressed in the user's coordinate system, not in device coordinates. The xgl_image() operator displays pixel data starting at a given reference point in the user's Model Coordinates. In the case of 3D Context, xgl_image() can perform Z-buffering.

For more information about raster primitives and attributes, see the *XGL Reference Manual.*

Raster Primitive Example Programs

The following programs illustrate raster-level primitives. The program below shows how the `xgl_context_set_pixel()` and `xgl_context_get_pixel()` primitives are used. The program `copy_raster.c` on page 217 provides an example of the `xgl_context_copy_buffer()` primitive.

Example Program for Getting and Setting Pixels

Get and set pixel operations are performed in the example program, `gspixel.c`. This program sets a pixel at 10,10 from the upper left corner of the raster to yellow and then prints a message if the operation was successful. If the pixel is obscured, a message is printed. To compile this program, type `make gspixel` in the example program directory. The `make` command compiles `gspixel.c`, `gspixel_main.c`, and `ex_utils.c`. The latter two programs are listed in Appendix B.

Code Example 7-4 Example for Getting and Setting Pixels

```
/*
 * gspixel.c
 */
#include <xview/xview.h>
#include <xgl/xgl.h>

#include "ex.h"


void
gspixel (Xgl_object      ctx)
{
    int                  i,
                         obscured;
    Xgl_color            set_pixel_color,
                         get_pixel_color;
    Xgl_pt_i2d           pos;

    /*
     * set pixel at 10, 10 to index value YELLOW
     */
    pos.x = pos.y = 10;
```

```
        set_pixel_color = yellow_color;
        xgl_context_set_pixel (ctx, &pos, &set_pixel_color);

        /*
         * get pixel from same location and return flag
         */
        obscured = xgl_context_get_pixel (ctx, &pos, &get_pixel_color);
        if (obscured) {
            /*
             * the pixel we are interested in covered by another
             * window or something else thus the color index
             * value in get_pixel_color is most likely incorrect
             */
            printf ("Pixel obscured\n");
        }
        else {
            /*
             * the pixel we read is NOT obscured therefore
             * we check to see if the value is YELLOW
             */
            if (ex_color_type == XGL_COLOR_INDEX &&
                get_pixel_color.index != YELLOW_INDEX) {
                /*
                 * we shouldn't get into this part of the if
                 * because the pixel value should be YELLOW
                 */
                printf ("INDEX: Get pixel error\n");
            }
            else if (ex_color_type == XGL_COLOR_RGB &&
                get_pixel_color.rgb.r != 1. &&
                get_pixel_color.rgb.g != 1.) {
                /*
                 * pixel value from get pixel function is equal
                 * to pixel value used in set pixel function
                 */
                printf ("RGB: Get pixel error\n");
            }
            else {
                printf ("Got YELLOW\n");
            }
        }
    }
```

Example Program for Copying Pixels

This example program, `copy_raster.c`, copies pixels from a Memory Raster into a Context. To compile this program, type `make copy_raster` in the example program directory. The `make` command compiles `copy_raster.c`, `copy_raster_main.c`, and `ex_utils.c`.

Code Example 7-5 Example for Copying Pixels

```
/*
 * copy_raster.c
 */

#include <xview/xview.h>
#include <math.h>

#include "ex.h"

#define MR_WIDTH        32
#define MR_HEIGHT       32
#define ARC_RADIUS      16
#define DEG2RAD(x)      (((x) * 3.141592654) / 180.)

void
copy_raster (Xgl_object  ctx)
{
    double              sqrt (double);
    int                 i;

    Xgl_object          win_ras;   /* window raster object */

    Xgl_arc_list        multiarc;  /* Xgl arc data structure */
    Xgl_arc_f2d         arc;       /* Xgl 2D-float-arc data
                                    * structure */
    Xgl_pt_i2d          cp_ras_pos;/* position of copy raster */

    /*
     * memory rasters objects for various positions of mouth
     */
    Xgl_object          open_mem_ras;     /* Mouth fully open */
    Xgl_object          half_mem_ras;     /* Mouth half open */
    Xgl_object          closed_mem_ras;   /* Closed Mouth */

    Xgl_color_type      color_type = XGL_COLOR_INDEX;
```

```
Xgl_color            surf_front_color;

xgl_object_get (ctx, XGL_CTX_DEVICE, &win_ras);

/*
 * set context surface color to YELLOW, set the new frame
 * action to be "wait for vertical retrace" then "clear
 * the area of the raster object associated with the context",
 * and increase number of subdivisions
 * for arcs so they look nice.
 */
surf_front_color = yellow_color;
xgl_object_set (ctx,
         XGL_CTX_SURF_FRONT_COLOR, &surf_front_color,
         XGL_CTX_NEW_FRAME_ACTION,
         XGL_CTX_NEW_FRAME_VRETRACE | XGL_CTX_NEW_FRAME_CLEAR,
         XGL_CTX_CURVE_APPROX_VALUE, 15.0,
         NULL);
/*
 * create memory raster used for source in copy raster
 * mouth open memory raster
 */
if (ex_win_depth == 24) ex_win_depth = 32;
if (ex_win_depth == 32) color_type = XGL_COLOR_RGB;

open_mem_ras = xgl_object_create (sys_st, XGL_MEM_RAS, 0,
                           XGL_DEV_COLOR_TYPE, color_type,
                           XGL_RAS_DEPTH, ex_win_depth,
                           XGL_RAS_WIDTH, MR_WIDTH,
                           XGL_RAS_HEIGHT, MR_HEIGHT,
                           NULL);
if (!open_mem_ras) {
    printf ("not enough memory (open memory raster)\n");
    return;
}

/*
 * mouth half open memory raster
 */
half_mem_ras = xgl_object_create (sys_st, XGL_MEM_RAS, 0,
                           XGL_DEV_COLOR_TYPE, color_type,
                           XGL_RAS_DEPTH, ex_win_depth,
                           XGL_RAS_WIDTH, MR_WIDTH,
                           XGL_RAS_HEIGHT, MR_HEIGHT,
                           NULL);
```

```
if (!half_mem_ras) {
    xgl_object_destroy (open_mem_ras);
    printf ("not enough memory (half memory raster)\n");
    return;
}

/*
 * mouth almost closed open memory raster
 */
closed_mem_ras = xgl_object_create (sys_st, XGL_MEM_RAS, 0,
                            XGL_DEV_COLOR_TYPE, color_type,
                            XGL_RAS_DEPTH, ex_win_depth,
                            XGL_RAS_WIDTH, MR_WIDTH,
                            XGL_RAS_HEIGHT, MR_HEIGHT,
                            NULL);
if (!closed_mem_ras) {
    xgl_object_destroy (open_mem_ras);
    xgl_object_destroy (half_mem_ras);
    printf ("not enough memory (closed memory raster)\n");
    return;
}

/*
 * set context device to open mouth memory object then
 * clear the memory of the memory raster and draw open mouth
 */
xgl_object_set (ctx, XGL_CTX_DEVICE, open_mem_ras, NULL);
xgl_context_new_frame (ctx);

/*
 * set multi-arc data structure to:
 *          1. one arc
 *          2. arc type is float 2D
 *          3. no bounding box
 *          4. point to arc data structure
 */
multiarc.num_arcs = 1;
multiarc.type = XGL_MULTIARC_F2D;
multiarc.bbox = NULL;
multiarc.arcs.f2d = &arc;

/*
 * set arc draw values to:
 *          1. center of arc is center of memory raster
 *          2. flag is always false (no edges)
 *          3. radius
```

```
 */
arc.center.x = (float) ARC_RADIUS;
arc.center.y = (float) ARC_RADIUS;
arc.center.flag = FALSE;
arc.radius = (float) ARC_RADIUS;

/*
 * by default, xgl draws arcs in a clockwise direction
 * because y is down by default and xgl always draws
 * arcs in a right-handed fashion open mouth arc
 */
arc.start_angle = DEG2RAD (45.); /* xgl draws in radians */
arc.stop_angle = DEG2RAD (315.); /* I think in degrees */
xgl_multiarc (ctx, &multiarc);

/*
 * set context device to half mouth memory object then
 * clear memory of memory raster and draw half mouth
 */
xgl_object_set (ctx, XGL_CTX_DEVICE, half_mem_ras, NULL);
xgl_context_new_frame (ctx);

/*
 * half open mouth arc
 */
arc.start_angle = DEG2RAD (22.5); /* xgl draws in radians */
arc.stop_angle = DEG2RAD (337.5); /* I think in degrees */
xgl_multiarc (ctx, &multiarc);

/*
 * set context device to closed mouth memory object then
 * clear memory of memory raster and draw closed mouth
 */
xgl_object_set (ctx, XGL_CTX_DEVICE, closed_mem_ras, NULL);
xgl_context_new_frame (ctx);

/*
 * (virtually) closed open mouth arc
 */
arc.start_angle = DEG2RAD (2.);   /* xgl draws in radians */
arc.stop_angle = DEG2RAD (358.5); /* I think in degrees */
xgl_multiarc (ctx, &multiarc);

/*
 * set device to saved raster and copy memory rasters to
 * context then clear the display
```

```
         */
         xgl_object_set (ctx, XGL_CTX_DEVICE, win_ras, NULL);
         for (i = 0; i < 248; i += 4) {
             /*
              * clear the context raster area
              */
             xgl_context_new_frame (ctx);

             /*
              * compute position of source raster in
              * destination raster using an interesting path
              */
             cp_ras_pos.x = i;
             cp_ras_pos.y = 50;

             /*
              * draw the three arc memory rasters and wait for
              * user to actually see the varying arc types
              */
             xgl_context_copy_buffer (ctx, NULL, &cp_ras_pos,
                                      open_mem_ras);
             sleep(0.5);

             xgl_context_copy_buffer (ctx, NULL, &cp_ras_pos,
                                      half_mem_ras);
             sleep(0.5);

            xgl_context_copy_buffer (ctx, NULL, &cp_ras_pos,
                                      closed_mem_ras);
        }

        xgl_context_new_frame (ctx);
        xgl_context_copy_buffer (ctx, NULL, &cp_ras_pos,
                                 closed_mem_ras);

        /*
         * restore context device object then destroy
         * memory raster object
         */

        xgl_object_set (ctx, XGL_CTX_DEVICE, win_ras, NULL);
        xgl_object_destroy (open_mem_ras);
        xgl_object_destroy (half_mem_ras);
        xgl_object_destroy (closed_mem_ras);
}
```

Accumulation Buffer

The XGL Raster object provides antialiasing of images with an accumulation buffer. The accumulation process takes a series of images and combines them using specified weights. Antialiasing by accumulating images in a buffer is not a real-time operation, but the quality of antialiasing can be seen to improve over time.

The accumulation buffer is a buffer containing an array of pixels where the contents of the accumulation process is stored. As an initial step in the accumulation process, the XGL application calls a series of XGL surface primitives and transfers the resulting image to the accumulation buffer. The application then changes the jitter offset and calls the same surface primitives again. Each rendering pass is jittered (moved) a subpixel distance from the previous pass and accumulated. The more passes that occur, the better the final image. The accumulation process involves weighted addition of the image buffer to the contents of the accumulation buffer. When slightly different images are averaged together, the resulting image is effectively antialiased. Typically, the application program accumulates into the accumulation buffer multiple times to produce an antialiased image and then copies the image to the display buffer. The process is complete when the image has an acceptable visual appearance.

The jitter offsets are determined by the application. The number of cycles for an acceptable image may be 60 or more for a 1×1 convolution filter (in other words, no convolution filter is applied). For some 3×3 convolution filters, eight cycles may be enough. Some hardware may have support for fast rendering but no support for convolution filtering; other hardware may support convolution filtering. On most hardware devices, a 1×1 convolution filter (no filtering) is used in accumulation. Please check your frame buffer documentation to determine whether a filter size other than 1×1 is supported.

The accumulation buffer enables an application to antialiase filled surfaces. It can also be used to render motion blur, depth of field, or soft shadows. Only one accumulation buffer is associated with an XGL Raster and, similarly, only one Raster is associated with an accumulation buffer. An accumulation buffer has the same width and height as the associated raster's image buffer.

Note – Accumulation buffer functionality is currently only available on RGB rasters.

Accumulation Operator

The operator xgl_context_accumulate() accumulates pixel information from the draw buffer of a raster to the destination buffer of the raster as specified by the XGL_3D_CTX_ACCUM_OP_DEST attribute. When the accumulation operation is complete, the contents of the accumulation buffer can optionally be copied to an image buffer.

The accumulation calculation is:

*(destination buffer) = (source buffer * source weight + destination buffer * accum weight)*

The source buffer is always the Context's draw buffer, and the destination buffer is determined by the attribute XGL_3D_CTX_ACCUM_OP_DEST. The weights are specified as input parameters to the operator.

The accumulation buffer is created in the raster when XGL_3D_CTX_ACCUM_OP_DEST is set to XGL_BUFFER_SEL_ACCUM from one of the Context's to which the raster is attached and the first time either xgl_context_accumulate() or xgl_context_clear_accumulation() is called. The accumulation buffer is destroyed when the raster is destroyed.

This accumulation operator is defined as:

```
void xgl_context_accumulate (
    Xgl_ctx             ctx,
    Xgl_bounds_i2d      *rectangle,
    Xgl_pt_i2d          *pos,
    float               src_wt,
    float               accum_wt,
    Xgl_buffer_sel      buf);
```

The arguments are defined as follows:

- ctx specifies the Context to which the source raster is attached and whose draw buffer is used as the source in the accumulation process. The destination buffer (specified by the attribute XGL_3D_CTX_ACCUM_OP_DEST) is used as the destination buffer in the accumulation operation.
- rectangle specified by xmin, xmax, ymin, ymax gives the source area of the draw buffer of the raster attached to the Context. If the value is NULL, the maximum area of the raster is used.

- pos specifies the position in the destination buffer to be used as the starting position. The width and height up to a maximum of width and height of the rectangle is accumulated. If any region exceeds the bounds of the raster, it is correspondingly clipped. If the value of NULL, the to and left corners of the raster are used.
- src_wt is the weight used as the source weight in the accumulation calculation above.
- accum_wt is the weight used as the accumulation weight in the accumulation calculation above.
- buf specifies the buffer that the accumulated image is copied to. buf can be set to one of the following:

  XGL_BUFFER_SEL_NONE
  XGL_BUFFER_SEL_DISPLAY
  A value of 0 to N-1, where N is the number of buffers allocated.

To copy the accumulated image, set buf to XGL_BUFFER_SEL_DISPLAY. When copying is specified, it is done from the accumulated buffer to the buffer specified by buf. The rectangle starting from position pos and the width and height up to a maximum of the width and height of the rectangle is the source area for copy. The destination area starts at {xmin,ymin} of the rectangle and extends to the maximum width and height specified by rectangle.

XGL_BUFFER_SEL_NONE specifies that no pixel data is copied back to the image buffer.

The accumulation buffer can be cleared using the xgl_context_clear_accumulation() operator. This operator clears the buffer specified by the attribute XGL_3D_CTX_ACCUM_OP_DEST to the value value. The operator is defined as:

```
xgl_context_clear_accumulation (
    Xgl_ctx          ctx,
    Xgl_color*       value);
```

The destination buffer can also be cleared by setting both the src_wt and the accum_wt to zero in the xgl_context_accumulate() operator.

Accumulation Attributes

The attribute XGL_3D_CTX_ACCUM_OP_DEST specifies the buffer that is used as the destination buffer during the accumulation operation and the clear operation. If the value of this operator is set to XGL_BUFFER_SEL_ACCUM, the accumulation buffer is used as the destination buffer for xgl_context_accumulate() and xgl_context_clear_accumulation(). The accumulation buffer is created the first time xgl_context_accumulate() or xgl_context_clear_accumulation() is called and XGL_3D_CTX_ACCUM_OP_DEST is set to XGL_BUFFER_SEL_ACCUM. If the value of XGL_3D_CTX_ACCUM_OP_DEST is set to XGL_BUFFER_SEL_DISPLAY, the current display buffer is used as the destination buffer for xgl_context_accumulate() and xgl_context_clear_accumulation(). The accumulation destination can also be set to XGL_BUFFER_SEL_NONE. The default value is XGL_BUFFER_SEL_NONE.

The attribute XGL_RAS_ACCUM_BUFFER_DEPTH is a read-only attribute identifying the depth of the accumulation buffer. The depth of the accumulation buffer is a multiple of the raster depth and is device-dependent. XGL_RAS_ACCUM_BUFFER_DEPTH can be XGL_ACCUM_DEPTH_1X, which implies that the depth of the accumulation buffer is at least the depth of the raster, or XGL_ACCUM_DEPTH_2X, which implies that the depth of the accumulation buffer is at least twice the depth of the raster.

The attribute XGL_3D_CTX_JITTER_OFFSET is a 3D Context attribute that specifies the amount by which the geometry should be offset in Device Coordinates before rendering. An application can use jitter offset to render an image before accumulating.

Accumulation Example

The following code fragment accumulates data into an accumulation buffer in double buffer mode. After the last pass, the accumulated image is copied to the display buffer.

Code Example 7-6 Accumulation Buffer Example

```
Xgl_object        win_ras;
Xgl_object        context;
Xgl_pt_d2d        jitter;
Xgl_accum_depth   depth;
Xgl_color         value;

win_ras = xgl_object_create(sys_st, XGL_WIN_RAS,&obj_desc, NULL);
context = xgl_object_create(sys_st, XGL_3D_CTX,  &obj_desc,
                  XGL_CTX_DEVICE, win_ras,
                  NULL);

/*
 * Set the destination to be the accumulation buffer
 */
xgl_object_set(context, XGL_3D_CTX_ACCUM_OP_DEST,
               XGL_BUFFER_SEL_ACCUM, 0);

/*
 *   The Jitter is default {0.0,0.0} on the context during the
 *   first rendering operation
 */
xgl_context_new_frame(context);
render_image(context);

/*
 * Clear the entire accumulation buffer to a value to zero
 */
value.rgb.r = 0.0;
value.rgb.g = 0.0;
value.rgb.b = 0.0;
xgl_context_clear_accumulation_buffer(context, value);

/*
* Set the Jitter offset and draw in to the draw buffer (buffer 1)
*/
jitter.x = -1./3.;
jitter.y = 0.;
xgl_object_set(context, XGL_3D_CTX_JITTER_OFFSET, &jitter, 0);
xgl_context_new_frame(context);
render_image(context);
xgl_context_accumulate(context, NULL, NULL, 0.50, 0.5,
              XGL_BUFFER_SEL_NONE);
```

```
jitter.x = 0.;
jitter.y = -1./3.;
xgl_object_set(context, XGL_3D_CTX_JITTER_OFFSET, &jitter, 0);
xgl_context_new_frame(context);
render_image(context);
xgl_context_accumulate(context, NULL, NULL, 1./3., 2./3.,
               XGL_BUFFER_SEL_NONE);

jitter.x = 1./3.;
jitter.y = 0.;
xgl_object_set(context, XGL_3D_CTX_JITTER_OFFSET, &jitter, 0);
xgl_context_new_frame(context);
render_image(context);
xgl_context_accumulate(context, NULL, NULL, 1./4., 3./4.,
              XGL_BUFFER_SEL_NONE);

jitter.x = 0.;
jitter.y = 1./3.;
xgl_object_set(context, XGL_3D_CTX_JITTER_OFFSET, &jitter, 0);
xgl_context_new_frame(context);
render_image(context);
xgl_context_accumulate(context, NULL, NULL, 1./5., 4./5.,
              XGL_BUFFER_SEL_NONE);

jitter.x = -1./3.;
jitter.y = -1./3.;
xgl_object_set(context, XGL_3D_CTX_JITTER_OFFSET, &jitter, 0);
xgl_context_new_frame(context);
render_image(context);
xgl_context_accumulate(context, NULL, NULL, 1./6., 5./6.,
              XGL_BUFFER_SEL_NONE);

jitter.x = -1./3.;
jitter.y = 1./3.;
xgl_object_set(context, XGL_3D_CTX_JITTER_OFFSET, &jitter, 0);
xgl_context_new_frame(context);
render_image(context);
xgl_context_accumulate(context, NULL, NULL, 1./7., 6./7.,
              XGL_BUFFER_SEL_NONE);

jitter.x = 1./3.;
jitter.y = -1./3.;
xgl_object_set(context, XGL_3D_CTX_JITTER_OFFSET, &jitter, 0)
xgl_context_new_frame(context);
render_image(context);
```

```
xgl_context_accumulate(context, NULL, NULL, 1./8., 7./8.,
              XGL_BUFFER_SEL_NONE);

/*
 * Accumulate and copy to display buffer (buffer 0)
 */
jitter.x = 1./3.;
jitter.y = 1./3.;
xgl_object_set(context, XGL_3D_CTX_JITTER_OFFSET, &jitter, 0);
xgl_context_new_frame(context);
render_image(context);
xgl_context_accumulate(context, NULL, NULL, 1./9., 8./9.,
              XGL_BUFFER_SEL_DISPLAY);
```

Rendering NURBS Curves and Surfaces with XGL

This chapter describes the XGL non-uniform rational B-spline (NURBS) curve and surface capabilities. After a brief introduction of NURBS and their applications, this chapter discusses the following:

- How to specify NURBS geometry.
- How to control the appearance of NURBS geometry.
- Examples and tips on how to use NURBS.

Introduction to NURBS Curves and Surfaces

NURBS curves and surfaces are concise, yet very powerful and general representations of a wide variety of simple and complex geometries. Trimmed NURBS are becoming increasingly more prominent in informal and formal industry standards for geometric modeling and computer graphics. This is primarily due to their generality and effectiveness for representing both simple and complex shapes, for their powerful mathematical properties, and for their ease of manipulation and processing on the computer. Typical applications range from mechanical CAD (computer-aided design) to scientific visualization.

NURBS curves and surfaces represent an exact mathematical definition of a curve or surface. They model a curve or surface exactly and can be displayed with any desired accuracy. The use of NURBS is appropriate for applications that require mathematically exact curves and surfaces; applications that do not require exact curves and surfaces may want to use other XGL curve and

surface primitives. While a polygonal representation of a surface is an approximation of the surface, polygonal surface primitives may be easier to use than NURBS, and they may have better performance than NURBS surfaces.

From a high-level mathematical description, a graphics system, such as XGL, tessellates curved geometry into lines (in the case of curves) or triangles (in the case of surfaces), whose sizes depend on the spatial relation of the curve or surface to the eye, and on the complexity of the geometry. This chapter discusses how this high-level description is specified with NURBS using XGL operators and attributes, and how the XGL application can control the display of NURBS curves and surfaces. XGL NURBS semantics follow the PHIGS PLUS model for NURBS and add additional features such as dynamic tessellation for trimming and speed hints for simple geometry.

NURBS Curves

A NURBS curve is a mapping from a bounded one-dimensional parameter space into a set of *m*-dimensional points in 2D or 3D object space. A NURBS curve is rendered as a 2D or 3D set of lines. Various attributes (in addition to line attributes) can be set to control the display of the curve.

Mathematical Description of a NURBS Curve

A NURBS curve is mathematically defined by:

1. A set of control points.
2. The order of the curve.
3. A knot vector defining the mapping from parameter space to object space.
4. A [min,max] range in parameter space that defines the portion of the curve that is displayed.

The curve is defined by:

$$C(t) = \sum_{i=1}^{n} B_{i,k}(t) P_i \qquad \text{(EQ 1)}$$

where

$C(t)$ is the curve;

P_i are the n control points;

t is the one-dimensional parameter;

$B_{i,k}(t)$ are the scalar-valued B-spline basis functions in the variable t, of order k (degree k-1).

The B-spline basis functions are defined by the order k and a knot vector, $\{t_j\}$ (j = 1 to $n+k$), where the sequence t_j is non-decreasing.

For more information on the mathematical characteristics of NURBS curves, refer to *Curves and Surfaces for Computer Aided Geometric Design* by Gerald Farin, Second Edition, Academic Press, Inc., San Diego, CA, 1990. The following sections discuss in more detail the three defining characteristics of a NURBS curve: control points, knot values, and parameter range.

Control Points

Control points determine the shape of the B-spline curve. The control points (P_i) of the curve may be defined using Cartesian coordinates (x,y,z) or using homogeneous coordinates (x,y,z,w). In the first case, the curve is said to be *non-rational*; in the second case, it is said to be *rational*.

The actual curve lies within the bounding polyhedron (called the *convex hull*) of the control points. When a control point is moved, the shape of the curve moves in that direction. This lends itself to interactive design. Figure 8-1 shows a B-spline curve of order 4 and its control points.

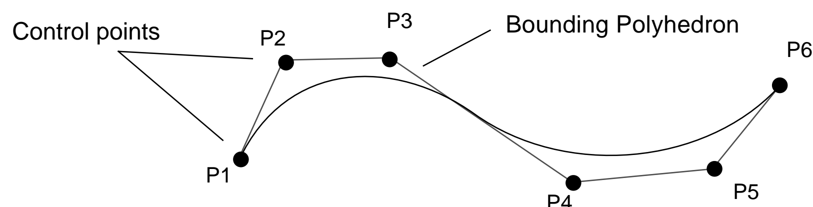


Figure 8-1 NURBS Curve Control Points and Bounding Polyhedron

Rational curves are more useful than non-rational curves in at least two ways. First, the homogeneous component (*w*) of each control point acts as a weight associated with it. The greater the weight, the more the curve is pulled toward that point. This characteristic offers greater interactive control over the shape of the curve. Second, simple geometry such as conics (circles, ellipses, etc.) can only be represented using rational curves. The example program `nurbs_circle.c` provides an example of representing circles as NURBS curves.

Knots and Parameter Range

A NURBS curve actually consists of a number of curve segments, each of which is defined by a polynomial curve defined by its Bezier coefficients. This characteristic of a NURBS curve provides local control over individual segments of the curve, since moving one control point of the B-spline affects only a small portion of the spline.

The knot sequence of the curve specifies the exact intervals in parameter space from which the curve segments are mapped into object space. A B-spline is said to be *uniform* if these intervals are all equal; otherwise, it is said to be *non-uniform*. The number of knots must equal the number of control points plus the order:

(number of knots) = (number of control points + order)

For example, a curve of order 4 with 6 control points may have the following uniform knot vector:

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

A curve with the following knot vector is non-uniform:

[0, 0, 0, 0, 1.5, 2, 2, 4, 4, 4, 4]

If the application only wants to create a Bezier curve of order *k*, it should use the knot sequence [0,0, ..*k* times, 1,1, ..*k* times].

A non-uniform knot sequence not only provides control over the curve's shape, but, by having multiple knots at the same value, it also provides control over the continuity between individual Bezier segments. For a NURBS curve of order 4, if there is only one knot at given value, there is second derivative continuity between the Bezier segments on either side. If there are two knots,

there is first order continuity. If there are three knots, there is positional continuity (the segments touch). If there are four knots, the curve is discontinuous at that point. Thus,

(degree of continuity at a knot) = (order - 1 - knot multiplicity)

The parameter range [umin,umax] specifies which portion of the curve is actually rendered. This is analogous to surface trimming. Figure 8-2 on page 233 shows a NURBS curve with a defined parameter range.

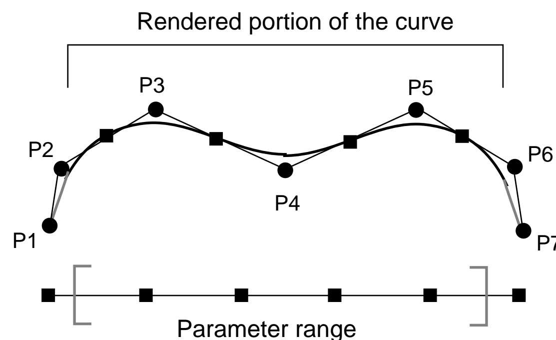


Figure 8-2 NURBS Curve Knot Vector and Parameter Range

Rendering a NURBS Curve

An XGL NURBS curve is rendered by calling `xgl_nurbs_curve()` or by creating a geometry cache (Gcache) from input data and rendering the Gcache every time the view or attributes change. Rendering NURBS curves with Gcaches will significantly improve performance. See Chapter 16, “Caching Geometry” for information on geometry caches for XGL NURBS.

The NURBS curve primitive generates a curve of the specified order based on the list of knots in the parameter space, the list of control points, and the parametric range. The NURBS curve primitive is:

```
void xgl_nurbs_curve (
   Xgl_ctx                ctx,
   Xgl_nurbs_curve        *curve,
   Xgl_bounds_f1d         *range,        /* optional */
   Xgl_curve_color_spline *color_spline) /* optional */
```

where `ctx` is the current Context, `curve` is a pointer to the definition of the curve, `range` is a pointer to a structure defining the parametric limits of the curve, and `color_spline` is a pointer to an *Xgl_curve_color_spline* structure that defines a curve whose control points are color coordinates. `range` and `color_spline` are optional and can be set to `NULL`.

The curve definition is defined in the *Xgl_nurbs_curve* structure as:

```
typedef struct {
     Xgl_usgn32          order;          /* Order of curve */
     Xgl_usgn32          num_knots;      /* Number of knots */
     float               *knot_vector; /* Knot vector */
     Xgl_pt_list         ctrl_pts; /* List of control points */
} Xgl_nurbs_curve;
```

In this structure, `order` is the curve order, which must be a positive integer. `num_knots` is the number of knots. `knot_vector` is a pointer to the knot vector. `ctrl_pts` is the array of control points. The control point coordinates are input as (*x,y,z*) for nonrational curves and (*x,y,z,w*) for rational curves.

The parameter `range` specifies which portion of the curve geometry is actually significant for display and picking. The range is defined in an *Xgl_bounds_f1d* structure:

```
typedef struct {
    float          bmin;
    float          bmax;
} Xgl_bounds_f1d;
```

The minimum range value should not be greater than the maximum value, and both values should lie between *knot_vector [order-1]* and *knot_vector [num_knots-order]*. If the range parameter is `NULL`, the whole curve is displayed.

NURBS Curve Attributes

Conceptually, NURBS curves are rendered by evaluating a number of sample points on the curve (also known as *tessellation*) and displaying straight lines between adjacent points. The greater the number of these points, the more accurate and smooth the rendering will be. However, tessellating a curve into a greater number of segments increases rendering time.

XGL allows the application to specify the smoothness of the rendered curve (in other words, how finely tessellated the curve is), thus determining performance. XGL provides the application with two attributes that control tessellation. The application can:

- Specify an *approximation type* that defines the criteria that is used for tessellating the curve into lines (XGL_CTX_NURBS_CURVE_APPROX).
- Specify how finely the curve should be tessellated (XGL_CTX_CURVE_APPROX_VAL).

Approximation types can be dynamic or static. Dynamic approximation lets the XGL graphics system decide on the tessellation, based on the spatial relation of the curve to the eye and the complexity of the geometry. This means that portions of the curve that are nearer to the eye and portions that have a more complicated geometry will be tessellated into more numerous, smaller lines than other portions. It also means that the tessellation may change as the application changes the view or modeling transforms. Static approximation defines a constant number of lines for the tessellation of the curve.

The attribute XGL_CTX_NURBS_CURVE_APPROX enables the application to specify the approximation type using the following values:

XGL_CURVE_CONST_PARAM_SUBDIV_BETWEEN_KNOTS
: This attribute is a *static* approximation type because it specifies a fixed number of lines into which the curve is tessellated, between any two pair of knots.

XGL_CURVE_METRIC_WC
XGL_CURVE_METRIC_VDC
XGL_CURVE_METRIC_DC
: A *metric* dynamic approximation type specifies an upper limit on the size of the lines generated. The values may apply to world coordinates (WC), virtual device coordinates (VDC), or device coordinates (DC).

XGL_CURVE_CHORDAL_DEVIATION_WC
XGL_CURVE_CHORDAL_DEVIATION_VDC
XGL_CURVE_CHORDAL_DEVIATION_DC
: A *chordal deviation* dynamic approximation type specifies an upper limit on the distance between the actual curve and the lines (chords) generated. The values may apply to world coordinates (WC), virtual device coordinates (VDC), or device coordinates (DC).

XGL_CURVE_RELATIVE_WC
XGL_CURVE_RELATIVE_VDC
XGL_CURVE_RELATIVE_DC

A *relative* dynamic approximation type indicates a relative quality of rendering quality to be maintained. The application can simply specify a value between 0 (lowest quality) to 1 (highest quality) for XGL_CTX_CURVE_APPROX_VAL, and let the XGL system figure out what metric and chordal deviation limits to use. The values may apply to world coordinates (WC), virtual device coordinates (VDC), or device coordinates (DC).

Figure 8-3 illustrates metric and chordal deviations.

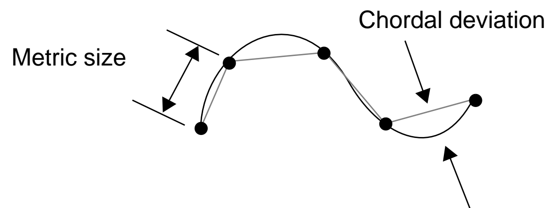


Figure 8-3 NURBS Curve Metric and Chordal Approximation Criteria

Note that even in the case of dynamic tessellation, the number of points generated may sometimes be too large for interactive rendering. The application can control the number of points used to display the curve with the attributes XGL_CTX_MIN_TESSELLATION and XGL_CTX_MAX_TESSELLATION. These attributes provide a lower and upper limit respectively on the number of points computed between a pair of knots on the curve.

XGL_CTX_CURVE_APPROX_VAL sets the approximation value for the curve. Refer to the *XGL Reference Manual* for information on how the approximation value relates to various approximation types, and for limitations on the order of the curve for these approximation types.

Color Splines

Color can be associated with a NURBS curve via the `xgl_nurbs_curve()` `color_spline` field, which points to an *Xgl_curve_color_spline* structure. This structure defines a spline whose control points are color coordinates. The color coordinates can be in 3D (nonrational) or 3D+w (rational). The color for every evaluated point on the geometry spline is determined by evaluating the RGB value at the corresponding parameter value on the color spline. This color is the vertex color for each evaluated point on the geometry spline. The Context line rendering attributes determine how the vertex colors are used during curve rendering.

Curve Example Programs

The following example programs show the use of NURBS curves. Each example is a fragment of a larger program. The complete program includes `nurbs_main.c` and `ex_utils.c`, both of which are listed in Appendix B, as well as all the example programs listed in this chapter. To compile the complete program, type `make nurbs` in the example program directory.

Bezier Curve Example Program

The following program, `nurbs_bezier.c`, displays a Bezier curve using NURBS.

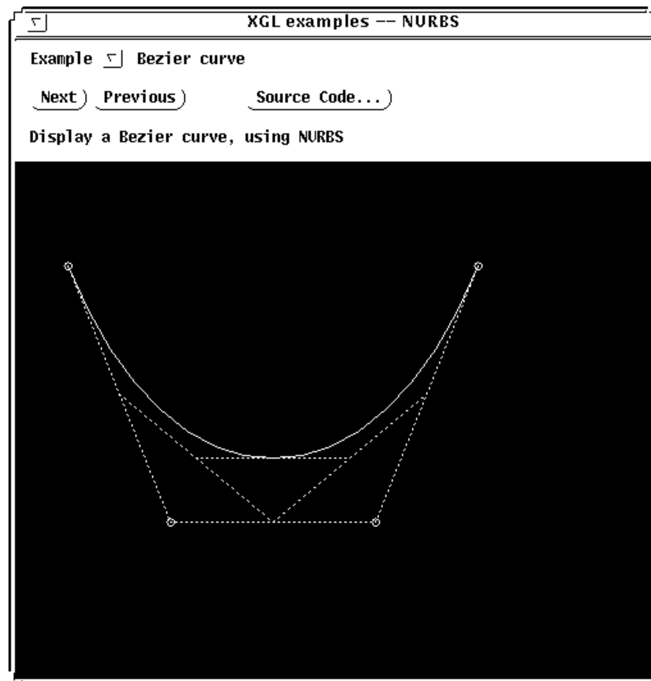


Figure 8-4 Output of `nurbs_bezier.c`

Code Example 8-1 Bezier Curve Example

```
/*
 * nurbs_bezier.c
 */

#include <xgl/xgl.h>
#include "ex.h"

void
nurbs_bezier (Xgl_object   ctx)
{

    Xgl_pt_list      pl;
    Xgl_pt_f2d       pt[20];
    Xgl_nurbs_curve curve;
    Xgl_bounds_f1d   range;
```

```
float           knot_v[20];
Xgl_color       color_m,
                color_l;

/*
 * clear the drawing area
 */

xgl_context_new_frame (ctx);

/*
 * set various attributes for the context
 */
color_m = white_color;
xgl_object_set (ctx, XGL_CTX_VDC_MAP, XGL_VDC_MAP_ASPECT,
       XGL_CTX_MARKER_SCALE_FACTOR, 6.0,
       XGL_CTX_MARKER_COLOR, &color_m,
       XGL_CTX_MARKER_DESCRIPTION, xgl_marker_circle,
       XGL_CTX_DEFERRAL_MODE, XGL_DEFER_ASAP,
       NULL);

/*
 * initialize the control points
 */

pt[0].x = -0.8;
pt[0].y = -0.6;
pt[1].x = -0.4;
pt[1].y = 0.4;
pt[2].x = 0.4;
pt[2].y = 0.4;
pt[3].x = 0.8;
pt[3].y = -0.6;

pl.num_pts = 4;
pl.pt_type = XGL_PT_F2D;
pl.pts.f2d = pt;
pl.bbox = (Xgl_bbox *) NULL;

/*
 * initialize a uniform knot vector to render
 * a bezier curve. Note that for a Bezier curve,
 * the knots are all the end-points.
 */
knot_v[0] = knot_v[1] = knot_v[2] = knot_v[3] = 0.0;
knot_v[4] = knot_v[5] = knot_v[6] = knot_v[7] = 1.0;
```

```
curve.num_knots = 8;
curve.knot_vector = knot_v;
curve.order = 4;
curve.ctrl_pts = pl;
range.bmin = 0.0;
range.bmax = 1.0;

/*
 * set the approximation criteria so that the rendered
 * curve should not deviate from the actual curve by more
 * than 1 pixel
 */

xgl_object_set (ctx,
    XGL_CTX_NURBS_CURVE_APPROX, XGL_CURVE_CHORDAL_DEVIATION_DC,
    XGL_CTX_NURBS_CURVE_APPROX_VAL, 1.0,
    NULL);

/*
 * draw the nurbs curve
 */

color_l = yellow_color;
xgl_object_set (ctx, XGL_CTX_LINE_COLOR, &color_l, NULL);

xgl_nurbs_curve (ctx, &curve, &range, NULL);

/*
 * draw a simple marker at each control point location.
 * Draw a polyline from the control points.
 */

color_l = green_color;
xgl_object_set (ctx,
      XGL_CTX_LINE_COLOR, &color_l,
      XGL_CTX_LINE_STYLE, XGL_LINE_PATTERNED,
      NULL);

xgl_multimarker (ctx, &pl);
xgl_multipolyline (ctx, (Xgl_bbox *) 0, 1, &pl);

/*
 * compute the subdivision of the bezier control points to
 * verify that the curve corresponds to a Bernstein polynomial.
 */
```

```
    pt[4].x = (pt[0].x + pt[1].x) / 2.0;
    pt[4].y = (pt[0].y + pt[1].y) / 2.0;

    pt[5].x = (pt[1].x + pt[2].x) / 2.0;
    pt[5].y = (pt[1].y + pt[2].y) / 2.0;

    pt[6].x = (pt[2].x + pt[3].x) / 2.0;
    pt[6].y = (pt[2].y + pt[3].y) / 2.0;

    color_l = cyan_color;
    xgl_object_set (ctx, XGL_CTX_LINE_COLOR, &color_l, 0);
    pl.pts.f2d = &(pt[4]);
    pl.num_pts = 3;
    xgl_multipolyline (ctx, (Xgl_bbox *) 0, 1, &pl);

    pt[7].x = (pt[4].x + pt[5].x) / 2.0;
    pt[7].y = (pt[4].y + pt[5].y) / 2.0;

    pt[8].x = (pt[5].x + pt[6].x) / 2.0;
    pt[8].y = (pt[5].y + pt[6].y) / 2.0;

    color_l = magenta_color;
    xgl_object_set (ctx, XGL_CTX_LINE_COLOR, &color_l, 0);
    pl.pts.f2d = &(pt[7]);
    pl.num_pts = 2;
    xgl_multipolyline (ctx, (Xgl_bbox *) 0, 1, &pl);

    xgl_object_set (ctx, XGL_CTX_LINE_STYLE, XGL_LINE_SOLID, NULL);

}
```

Circle Curve Example Program

The following program, `nurbs_circle.c`, displays circles as curves using different approximation criteria.

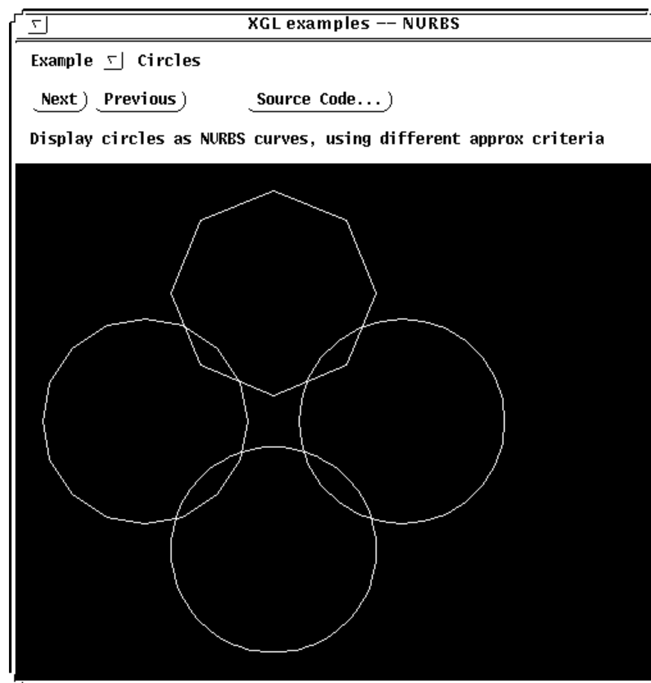


Figure 8-5 Output of `nurbs_circle.c`

Code Example 8-2 Circle Curve Example

```
/*
 * nurbs_circle.c
 */

#include <xgl/xgl.h>
#include <math.h>
#include "ex.h"
```

```
void            create_nurbs_circle (
                        float,
                        float,
                        float,
                        Xgl_nurbs_curve*,
                        Xgl_bounds_f1d*);

void
nurbs_circle (Xgl_object  ctx)
{

    Xgl_pt_list       pl;
    Xgl_pt_f2h        pt_w[20];
    Xgl_nurbs_curve   curve;
    Xgl_bounds_f1d    range;
    float             knot_v[20];
    Xgl_color         color_m,
                      color_l

    /*
     * clear the drawing area
     */

    xgl_context_new_frame (ctx);

    /*
     * set various attributes for the context
     */

    xgl_object_set (ctx, XGL_CTX_VDC_MAP, XGL_VDC_MAP_ASPECT,
                  XGL_CTX_DEFERRAL_MODE, XGL_DEFER_ASAP,
                  NULL);


    /*
     * draw four circles. for each:
     * 1. construct a circle of the same size at a different location
     * 2. use a different approximation criteria
     * 3. draw nurbs curve
     */

    pl.pt_type = XGL_PT_F2H;
    pl.pts.f2h = pt_w;
    curve.ctrl_pts = pl;
    curve.knot_vector = knot_v;
```

```
/*
 * I. constant # subdivisions between knots (static tessellation)
 */

color_l = green_color;
xgl_object_set (ctx,
        XGL_CTX_NURBS_CURVE_APPROX,
        XGL_CURVE_CONST_PARAM_SUBDIV_BETWEEN_KNOTS,
        XGL_CTX_NURBS_CURVE_APPROX_VAL, 7.0,
        XGL_CTX_LINE_COLOR, &color_l,
        NULL);

create_nurbs_circle (0.5, 0.0, 0.4, &curve, &range);
xgl_nurbs_curve (ctx, &curve, &range, NULL);

/*
 * II. relative "quality" measure (0 to 1) in DC.
 */

color_l = white_color;
xgl_object_set (ctx,
        XGL_CTX_NURBS_CURVE_APPROX,
        XGL_CURVE_RELATIVE_DC,
        XGL_CTX_NURBS_CURVE_APPROX_VAL, 0.9,
        XGL_CTX_LINE_COLOR, &color_l,
        NULL);

create_nurbs_circle (0.0, 0.5, 0.4, &curve, &range);
xgl_nurbs_curve (ctx, &curve, &range, NULL);

/*
 * III. limit on size (in VDC - 0 to 1) of line segments generated
 */

color_l = cyan_color;
xgl_object_set (ctx,
        XGL_CTX_NURBS_CURVE_APPROX,
        XGL_CURVE_METRIC_VDC,
        XGL_CTX_NURBS_CURVE_APPROX_VAL, 0.2,
        XGL_CTX_LINE_COLOR, &color_l,
        NULL);

create_nurbs_circle (-0.5, 0.0, 0.4, &curve, &range);
xgl_nurbs_curve (ctx, &curve, &range, NULL);
```

```
    /*
     * IV. limit on deviation (in WC) of approximated curve from
     * actual
     */

    color_l = yellow_color;
    xgl_object_set (ctx,
           XGL_CTX_NURBS_CURVE_APPROX,
           XGL_CURVE_CHORDAL_DEVIATION_WC,
           XGL_CTX_NURBS_CURVE_APPROX_VAL, 0.02,
           XGL_CTX_LINE_COLOR, &color_l,
           NULL);

    create_nurbs_circle (0.0, -0.5, 0.4, &curve, &range);
    xgl_nurbs_curve (ctx, &curve, &range, NULL);

 }

/*
 * given a center and radius of a circle, create nurbs control pts
 * and knots. Asummes dynamic storage is pre-allocated.
 */

void
create_nurbs_circle (
   float              center_x,
   float              center_y,     /* input  */
   float              radius,
   Xgl_nurbs_curve    *curve,       /* output */
   Xgl_bounds_f1d     *range)
{

    float isqrt_2 = sqrt ((double) 0.5);
    Xgl_pt_f2h   *cpt;
    float        *knot;

    /*
     * initialize the control points
     */

    curve->order = 3;
    curve->ctrl_pts.num_pts = 9;
    curve->ctrl_pts.pt_type = XGL_PT_F2H;
    cpt = curve->ctrl_pts.pts.f2h;

    cpt[0].x = center_x + radius;
```

```
    cpt[0].y = center_y;
    cpt[0].w = 1.0;
    cpt[1].x = (center_x + radius) * isqrt_2;
    cpt[1].y = (center_y + radius) * isqrt_2;
    cpt[1].w = isqrt_2;
    cpt[2].x = center_x;
    cpt[2].y = center_y + radius;
    cpt[2].w = 1.0;
    cpt[3].x = (center_x - radius) * isqrt_2;
    cpt[3].y = (center_y + radius) * isqrt_2;
    cpt[3].w = isqrt_2;
    cpt[4].x = center_x - radius;
    cpt[4].y = center_y;
    cpt[4].w = 1.0;
    cpt[5].x = (center_x - radius) * isqrt_2;
    cpt[5].y = (center_y - radius) * isqrt_2;
    cpt[5].w = isqrt_2;
    cpt[6].x = center_x;
    cpt[6].y = center_y - radius;
    cpt[6].w = 1.0;
    cpt[7].x = (center_x + radius) * isqrt_2;
    cpt[7].y = (center_y - radius) * isqrt_2;
    cpt[7].w = isqrt_2;
    cpt[8].x = center_x + radius;
    cpt[8].y = center_y;
    cpt[8].w = 1.0;

    /*
     * use a nonuniform knot vector to render a circle.
     * recall the relation: # knots = # ctrl pts + order
     */

    curve->num_knots = 12;
    knot = curve->knot_vector;

    knot[0] = knot[1] = knot[2] = 0.;
    knot[3] = knot[4] = 1.;
    knot[5] = knot[6] = 2.;
    knot[7] = knot[8] = 3.;
    knot[9] = knot[10] = knot[11] = 4.;

    range->bmin = 0.;
    range->bmax = 4.;
}
```

NURBS Surfaces

A NURBS surface is a mapping from two-dimensional parameter space into object space. The NURBS surface is rendered as a set of triangles. Various attributes can be set to control the display of the surface.

Note – Some of the geometry and attributes for NURBS surfaces are logical extensions of NURBS curve geometry and attributes. This section assumes that you have read the section on NURBS curves; therefore, not all of the information covered in that section is repeated here.

Mathematical Description of a NURBS Surface

A NURBS surface is mathematically defined by:

1. A two-dimensional grid of control points.
2. The order of the surface in *u* and *v*.
3. Knot sequences in *u* and *v* (the two dimensions of the parameter space).
4. Trimming information that specifies which portion of the rectangular parameter space is actually mapped to object space.

The surface is defined by:

$$S(u, v) = \sum_{i=1}^{n} \sum_{j=1}^{m} B_{i,k}(u) B_{j,l}(v) P_{i,j} \qquad \text{(EQ 2)}$$

where

$S(u,v)$ is the surface;

$P_{i,j}$ are an $(n \times m)$ array of control points;

u and v are the two parameters;

$B_{i,k}(u)$ is the i-th B-spline basis function of order k, defined by the knot vector $\{u_p\}$ ($p = 1$ to $n+k$);

$B_{j,l}(v)$ is the *j*-th B-spline basis function of order *l*, defined by the knot vector $\{v_q\}$ $(q = 1$ to $m{+}l)$

The B-spline basis functions are defined by the orders *k* and *l*, and their respective non-decreasing knot sequences.

For more information on the mathematics of B-spline surfaces, refer to *Curves and Surfaces for Computer Aided Geometric Design* by Gerald Farin, Second Edition, Academic Press, Inc., San Diego, CA, 1990. The following sections discuss surface control points and knot vectors.

Control Points

The above mathematical definition of a NURBS surface is best understood by considering it as an extension to the definition of a NURBS curve, since the inner loop of equation EQ2 is simply the definition of a NURBS curve. If you start with a NURBS curve and sweep the curve in an orthogonal direction by defining a NURBS curve path for each control point, the path swept by the original curve defines the surface.

The control points ($P_{i,j}$) may be defined in Cartesian or homogeneous coordinates. Accordingly, the surface is said to be *non-rational* or *rational*. As in the case of curves, the actual surface follows the shape of the control polygon. Simple geometry such as planar or conics (spherical, cylindrical) can only be represented using rational surfaces.

Knots

Just as a NURBS curve consists of a number of curve segments, a NURBS surface consists of surface segments called *patches*. The *u* and *v* knot sequences define a grid in two-dimensional parameter space and specify the exact intervals in parameter space from which the segments are mapped onto object space. A B-spline is said to be uniform if all *u* knot intervals are equal, and all *v* knot intervals are equal; otherwise, it is said to be non-uniform. Refer to page 232 for more details on knot vectors.

Rendering a NURBS Surface

A NURBS surface is specified and rendered by calling `xgl_nurbs_surface()` or by creating a geometry cache (Gcache) from input data and rendering the Gcache every time the view or attributes changes. Using a Gcache for rendering will significantly increase performance. See Chapter 16, "Caching Geometry" for information on geometry caches for XGL NURBS.

The NURBS surface primitive is:

```
void xgl_nurbs_surface (
   Xgl_ctx                    ctx,
   Xgl_nurbs_surf             *surface,
   Xgl_trim_loop_list         *trimming,      /* optional */
   Xgl_nurbs_surf_simple_geom *hints,         /* optional */
   Xgl_surf_color_spline      *color_spline,  /* optional*/
   Xgl_surf_data_spline_list  *data_splines) /* not implemented */
```

In this structure, `ctx` is the current Context, `surface` is a pointer to the definition of the surface, `trimming` is a pointer to a structure defining the surface trimming information, and `hints` is a pointer to a structure providing information about the shape of the surface. `color_spline` is a pointer to a NURBS surface whose control points are color coordinates.

The *Xgl_nurbs_surf* structure is defined as:

```
typedef struct {
     Xgl_usgn32       order_u;       /* Order of surface in u */
     Xgl_usgn32       order_v;       /* Order of surface in v */
     Xgl_usgn32       num_knots_u;   /* Number of knots in u */
     float            *knot_vector_u;/* Knot vector in u */
     Xgl_usgn32       num_knots_v;   /* Number of knots in v */
     float            *knot_vector_v;/* Knot vector in v */
     Xgl_pt_list      ctrl_pts;   /* Row-ordered list of control
                                  points (index changes faster in
                                  u than v) */
 } Xgl_nurbs_surf;
```

This structure gives the order of the surface, the surface *u* and *v* directions, the number of knots in the *u* and *v* parameters, the knot vectors in the *u* and *v* directions, and the array of control points for the surface.

Surface Trimming

B-spline surface mapping is defined on a two-dimensional parameter space rectangle defined by the knot vectors. Figure 8-6 on page 251 shows an untrimmed surface and the grid of control points.

Not all real-life surfaces have a rectangular topology. To define a non-rectangular surface, a set of *trimming loops* can be specified to carve out portions of the rectangular region. Only the portion enclosed by trimming loops is significant (rendered), and the rest of the surface is discarded. If no trimming is specified, the entire rectangular region is displayed.

Note – Surface trimming is limited to NURBS surfaces with an order of 1 through 9 in either the *u* or *v* direction.

Each trimming loop consists of one or more *trimming curves*. Trimming curves are NURBS curves in 2D parameter space joined in a head-to-tail fashion. The trimming loop should be continuous, closed, and should not intersect itself or another trimming loop. In general, an individual trimming curve on a trimming loop should only touch another trimming curve or itself when they are adjacent in a trimming loop, and then only at their endpoints. However, XGL's NURBS implementation allows trimming curves to touch at isolated points as long as they do not coincide along any finite length.

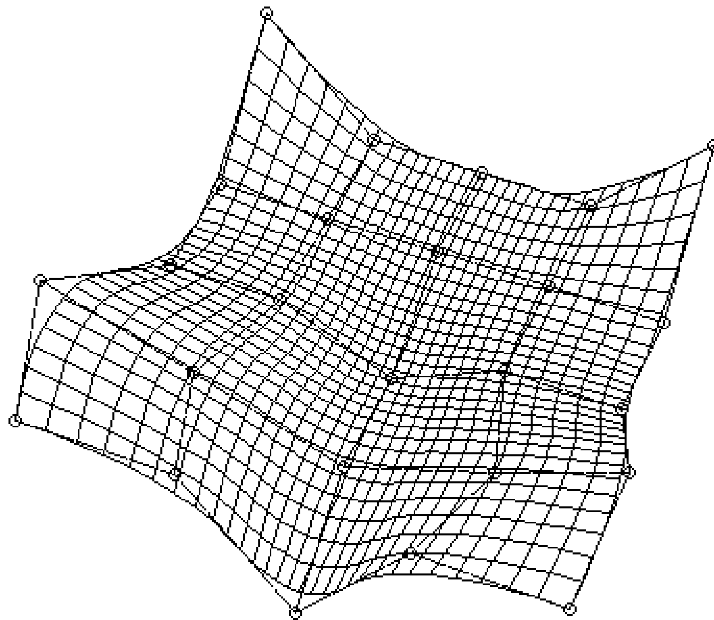


Figure 8-6 Untrimmed NURBS Surface

Trimming curves define a mapping from a one-dimensional parameter space to the two-dimensional parameter space of the surface. Refer to the section on NURBS curves for the definition and properties of curves used as trimming curves. Trimming curves should have an order greater than 1. Note that surfaces with an order above nine in either the *u* or *v* direction cannot be trimmed.

The significant region of the trimmed surface is doubly defined through the following rules:

1. Odd Winding Rule: A point is in the significant region if a ray projected from that point in any direction has an odd number of intersections with the set of trimming loops.
2. Curve Orientation Rule: For any given directed trimming loop, the significant region is “to the left”, and the outside region is “to the right”. What this means is that if the trimming curve is in a clockwise direction, the region of the surface inside the trimming curve is not rendered. If the

trimming curve is counterclockwise, the region of the surface inside the trimming curve is rendered. For the definition of the terms “to the left” and “to the right”, see the PHIGS PLUS documentation.

Figure 8-7 shows trimming loops in the surface’s parameter space.

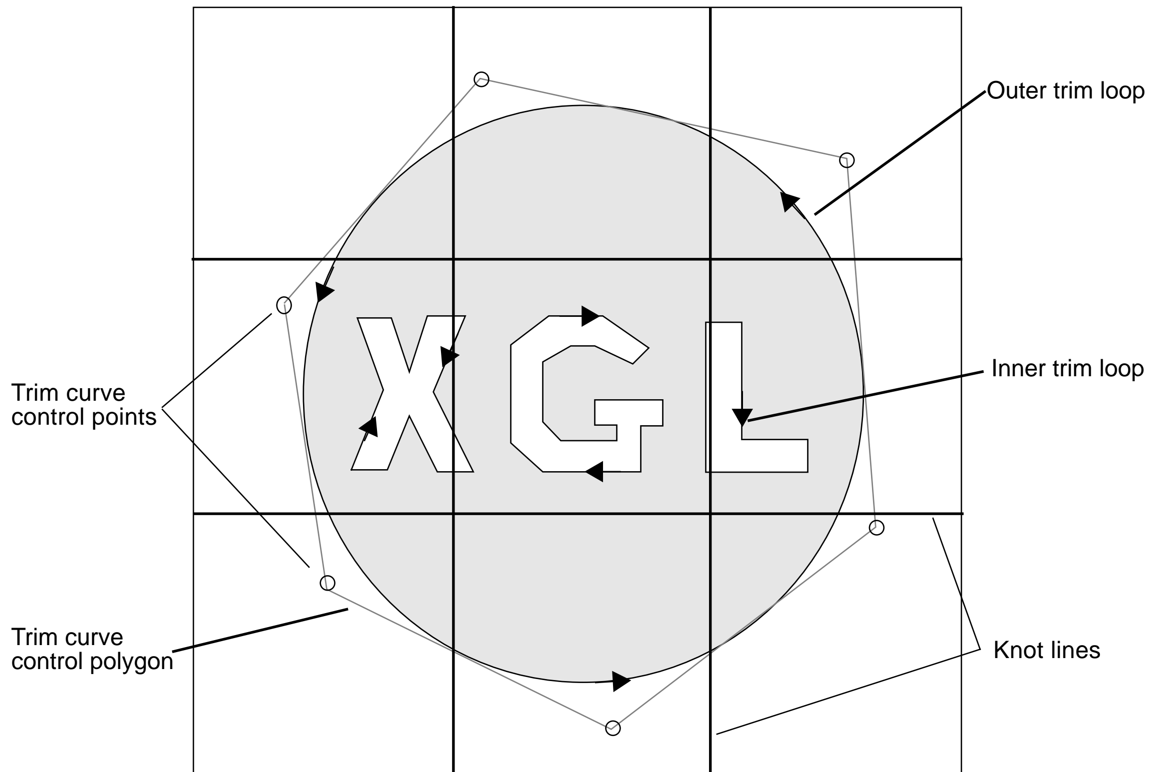


Figure 8-7 Trimming Curves in Surface Parameter Space

The NURBS implementation handles wrong trimming input by flagging errors and using a “recover and display as much as possible” philosophy. If a trimming loop is not continuous, the trimming curves are adjusted to close the loop. XGL attempts to localize other trimming input errors by displaying as much of the trimmed surface as possible and treating the “wrong” portions as untrimmed.

Figure 8-8 on page 253 shows a trimmed surface.

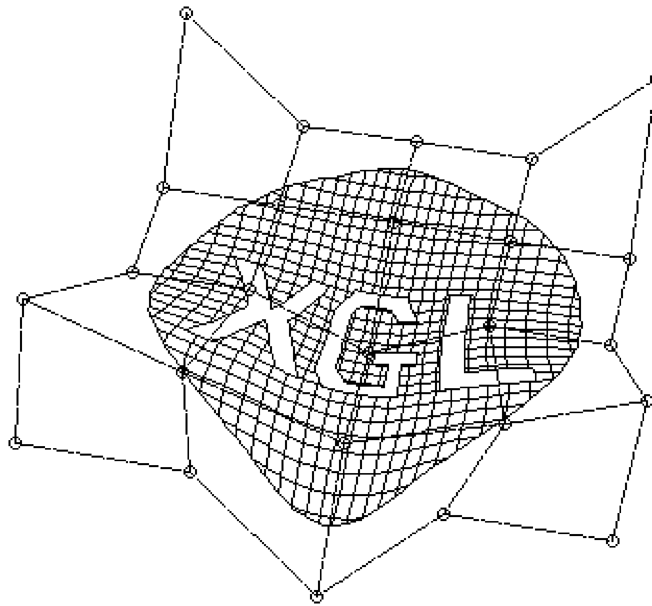


Figure 8-8 Trimmed NURBS Surface

Performance Hints for Simple Geometry

A majority of real-life surfaces, especially in mechanical CAD, are fairly simple. They may be just planar, or they may be conics (spherical, cylindrical, conical). For simple surfaces, the application may want to use one of the polygonal XGL surface primitives, such as `xgl_polygon()`, since these primitives are easier to define and may have better performance. In some cases, however, the application may want to use NURBS to represent surfaces because of the generality of NURBS and the ability to trim these surfaces. For example, a trimmed NURBS representation is well suited to represent the surfaces of the Boolean intersection of a cylinder and a cone.

Even when the NURBS representation is adequate for the display of simple surfaces, the graphical processing (in particular, the computation of surface normals for lighting) can be speeded up considerably if the application lets XGL know about the original simple geometry. For example, if the XGL

software knows that a given NURBS surface is spherical, it can compute the normal at a given point on the surface by simply taking the vector from the center of the sphere to the given point, instead of taking the cross product of the derivative surface functions. This can speed up display substantially.

Thus, in addition to giving the control points, knot vectors, and trimming information, the application can optionally provide hints about the original geometry via the `hints` field of the `xgl_nurbs_surface()` primitive. This field points to an *Xgl_nurbs_surf_simple_geom* structure that provides information on the type of surface and the original geometry. Table 8-1 lists the available surface types and the geometry required.

Table 8-1 Providing Hints for NURBS Surface Geometry

Surface	Geometry Hints
Planar	Surface normal
Cylindrical	Point on axis, axis direction, radius, flag indicating the front and back sides of the surface
Conical	Apex, axis direction, cone angle, flag indicating the front and back sides of the surface
Spherical	Center, radius, flag indicating the front and back sides of the surface

If the geometry hints are inconsistent with the NURBS information, the surface may be shaded wrongly. Please refer to the *XGL Reference Manual* for information on the data structures used for surface hints.

NURBS Surface Attributes

Just as NURBS curves are rendered as a set of lines, NURBS surfaces are rendered by evaluating the surface at a number of points and generating triangles. XGL enables the application to control the smoothness of the surface tessellation using the following attributes:

- `XGL_CTX_NURBS_SURF_APPROX`, which specifies the criteria uses for tessellation.
- `XGL_CTX_NURBS_SURF_APPROX_VAL_U` and `XGL_CTX_NURBS_SURF_APPROX_VAL_V`, which supply *u* and *v* values for tessellation.

XGL_CTX_NURBS_SURF_APPROX gives the application a choice of the following surface approximation types:

XGL_SURF_CONST_PARAM_SUBDIV_BETWEEN_KNOTS
A *static* approximation type analogous to the curve approximation type XGL_CURVE_CONST_PARAM_SUBDIV_BETWEEN_KNOTS.

XGL_SURF_METRIC_WC
XGL_SURF_METRIC_VDC
XGL_SURF_METRIC_DC
A *dynamic* approximation type analogous to the curve metric approximation types. The values may apply to world coordinates (WC), virtual device coordinates (VDC), or device coordinates (DC).

XGL_SURF_CHORDAL_DEVIATION_WC
XGL_SURF_CHORDAL_DEVIATION_VDC
XGL_SURF_CHORDAL_DEVIATION_DC
A *dynamic* approximation type analogous to the curve chordal deviation approximation types. The values may apply to world coordinates (WC), virtual device coordinates (VDC), or device coordinates (DC).

XGL_SURF_RELATIVE_WC
XGL_SURF_RELATIVE_VDC
XGL_SURF_RELATIVE_DC
A *dynamic* approximation type analogous to the curve relative approximation types. The values may apply to world coordinates (WC), virtual device coordinates (VDC), or device coordinates (DC).

XGL_CTX_NURBS_SURF_APPROX_VAL_{U,V} sets the approximation quality of NURBS surfaces as follows:

- For constant parametric approximation, the attribute represents the number of subdivisions of the surface.
- For metric approximation, the attribute represents a maximum distance between two points in the *u* or *v* parametric directions.
- For chordal deviation approximation, the attribute sets the maximum deviation between the surface and the approximation. Only XGL_CTX_NURBS_SURF_APPROX_VAL_U is used.
- For relative approximations, the attribute sets a relative quality measure. Only XGL_CTX_NURBS_SURF_APPROX_VAL_U is used

As in the case of curves, the application can specify limits on the number of sample points generated between two consecutive *u/v* knot values using `XGL_CTX_MIN_TESSELLATION` and `XGL_CTX_MAX_TESSELLATION`.

Refer to page 234 for the descriptions of the curve approximation types. Refer to the *XGL Reference Manual* for more information on how the approximation values relate to the approximation types.

Controlling the Appearance of Trimming Curves

The approximation criteria for trimming curves are specified as part of the trimming curve geometry itself so that individual trimming curves may have their own properties. A trimming curve is defined as follows:

```
typedef struct {
/* control points, knot vector, parameter range: same as for
          NURBS curves */
   Xgl_usgn32             order;
   Xgl_usgn32             num_knots;
   float                  *knot_vector;
   Xgl_bounds_f1d         range;
   Xgl_pt_list            ctrl_pts;
/*
* trimming curve structure fields
*/
   Xgl_boolean            trim_curve_vis; /* Highlight trim curve*/
   Xgl_trim_curve_approx  trim_curve_approx;   /* Approx type */
   float                  trim_curve_approx_value; /*Approx
value*/
} Xgl_trim_curve;
```

The *Xgl_trim_curve_approx* structure is defined as:

```
typedef enum {
   XGL_TRIM_CURVE_CONST_PARAM_SUBDIV_BETWEEN_KNOTS;
                                          /* static tessellation */
   XGL_TRIM_CURVE_VIEW_DEPENDENT;      /* dynamic tessellation */
} Xgl_trim_curve_approx;
```

In the case of XGL_TRIM_CURVE_CONST_PARAM_SUBDIV_BETWEEN_KNOTS, a fixed number of points to be sampled between two knot values of the trimming curve is given by the field trim_curve_approx_value.

XGL_TRIM_CURVE_VIEW_DEPENDENT is a dynamic approximation type that allows XGL to decide on the number of sample points for a curve by considering the tessellation of the portion of the surface it lies on and the geometry of the curve itself. In this case, the trim_curve_approx_value field should provide a number between 0 and 1. This number is mapped to a size threshold in parameter space. A step size is computed for the trim curve using this threshold as well as the subdivisions of the portion of the surface it lies on. This ensures that the trimmed surface is tessellated well in cases when the trimming curve geometry is more complicated than the underlying surface, or vice versa.

The trimming curves constitute the boundaries of the surface. If the surface is not trimmed, the parametric limits constitute the boundaries. These boundaries are rendered using edge attributes. The trim_curve_vis flag is used to control the visibility of individual trim curves. If the front/back surface fill style (as defined by XGL_3D_CTX_SURF_BACK_FILL_STYLE or XGL_CTX_SURF_FRONT_FILL_STYLE) is XGL_SURF_FILL_HOLLOW, the surface is rendered only along these surface boundaries.

NURBS Surface Parametric Style

In addition to specifying the interior surface fill style and trimming edges, XGL allows the application to enhance the display of the NURBS surface using the XGL_CTX_NURBS_SURF_PARAM_STYLE attribute. This attribute can take the following values:

XGL_SURF_PLAIN
: No additional lines or edges are displayed on the surface. This is the default value.

XGL_SURF_ISO_CURVES
: Isoparametric curves are drawn on the surface. This attribute allows the application to specify that a fixed number of lines be drawn over the surface to give a wireframe appearance. The placement of the isoparametric curves is specified with the attribute XGL_CTX_NURBS_SURF_ISO_PLACEMENT, which can take the following values:

- XGL_ISO_CURVE_BETWEEN_KNOTS: Lines are uniformly spaced between two consecutive knots.
- XGL_ISO_CURVE_BETWEEN_LIMITS: Lines are uniformly spaced over the entire surface.

The number of lines in *u* and *v* directions is specified with the attributes XGL_CTX_NURBS_SURF_ISO_CURVE_U_NUM and XGL_CTX_NURBS_SURF_ISO_CURVE_V_NUM. These lines have polyline attributes and are rendered on top of the shaded surface (if the latter is displayed), but they have lower priority than the edges of the surface.

XGL_SURF_SHOW_TESSELLATION
Edges of all tessellated facets are drawn using edge attributes. This is useful in visualizing the approximated surface.

XGL_SURF_INCR_SILHOUETTE_TESS
For better rendering quality, the surface is tessellated more finely near areas of silhouette edges. This value is valid only for dynamic surface approximation types and is ignored for static approximation types.

XGL_SURF_DEVICE_DEPENDENT
Same as XGL_SURF_PLAIN. No additional lines or edges are rendered.

A combination that is often used to implement wireframe mode is to turn off lighting, turn on edges, and display isoparametric lines.

Color Surfaces

Color can be associated with a NURBS surface via the xgl_nurbs_surface() color_spline field, which points to an *Xgl_surf_color_spline* structure. This structure defines a surface whose control points are color coordinates. The color coordinates can be in 3D (nonrational) or 3D+w (rational). The color for every evaluated point on the geometry surface is determined by evaluating the RGB value at the corresponding parameter value on the color surface. This color is the vertex color for each evaluated point on the geometry surface. The Context surface illumination attributes determine how the vertex colors are used during surface rendering.

If the color spline data is not provided for a surface, surface front/back color is used instead, and surface illumination is based only on intrinsic color.

Displaying Silhouettes on NURBS Surfaces

An application can display the silhouette edges between visible and hidden portions of the surface by setting the XGL_3D_CTX_SURF_SILHOUETTE_EDGE_FLAG attribute in the Context to TRUE. Because the human eye is sensitive to areas near silhouette edges, finer tessellation in these areas may be needed. The XGL_SURF_INCR_SILHOUETTE_TESS value of the attribute XGL_CTX_NURBS_SURF_PARAM_STYLE can be used to increase the tessellation near silhouette edges. Note that setting this value does not have any effect for static approximation types.

The XGL_3D_CTX_SURF_SILHOUETTE_EDGE_FLAG and XGL_SURF_INCR_SILHOUETTE_TESS attributes are independent of each other. If XGL_SURF_INCR_SILHOUETTE_TESS is set, but XGL_3D_CTX_SURF_SILHOUETTE_EDGE_FLAG is not set, areas near silhouette edges will be tessellated more finely, but the silhouette edges themselves will not be displayed.

If the surface has discontinuous first derivatives in either the *u* or *v* directions, silhouette edges may be displayed incorrectly. If the multiplicity of each internal knot is less than or equal to the order of the surface minus 2 in each direction, the surface will always have continuous first derivatives.

Surface Example Program

The following example program shows a trimmed sphere. The example is a fragment of a larger program, which includes nurbs_main.c and ex_utils.c, both of which are listed in Appendix B, as well as all the example programs listed in this chapter. To compile the complete program, type make nurbs in the example program directory.

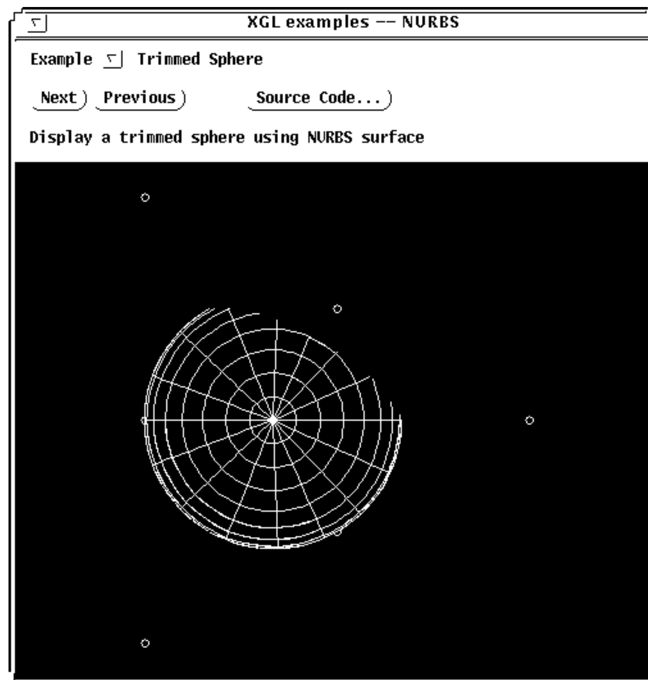


Figure 8-9 Output of `nurbs_sphere.c`

Code Example 8-3 NURBS Surface Example

```
/*
 * nurbs_sphere.c
 */

#include <xgl/xgl.h>
#include "ex.h"

/* data values for surface parameters */

/* knot vectors */

static float     knot_vector_u[10]     = {-1,-1,-1,0,0,1,1,2,2,2};
static float     knot_vector_v[8]      = {-1,-1,-1,0,0,1,1,1};

/* control points for sphere centered at origin with radius 0.5 */
```

```
static Xgl_pt_f3h ctrl_pts_arr[35]      = {

   { 0,          0,          -1.41421,          2.82843 },
   { 0,          0,          -0.707107,         1.41421 },
   { 0,          0,          -1.41421,          2.82843 },
   { 0,          0,          -0.707107,         1.41421 },
   { 0,          0,          -1.41421,          2.82843 },
   { 0,          0,          -0.707107,         1.41421 },
   { 0,          0,          -1.41421,          2.82843 },

   { -1.0,       0,          -1.0,              2       },
   { -0.5,       -0.866025,  -0.5,              1       },
   { 0.5,        -0.866025,  -1.0,              2       },
   { 1.0,        0,          -0.5,              1       },
   { 0.5,        0.866025,   -1.0,              2       },
   { -0.5,       0.866025,   -0.5,              1       },
   { -1.0,       0,          -1.0,              2       },

   {-1.41421,    0,          0,                 2.82843 },
   {-0.707107,   -1.22474,   0,                 1.41421 },
   { 0.707107,   -1.22474,   0,                 2.82843 },
   { 1.41421,    0,          0,                 1.41421 },
   { 0.707107,   1.22474,    0,                 2.82843 },
   { -0.707107,  1.22474,    0,                 1.41421 },
   { -1.41421,   0,          0,                 2.82843 },

   { -1.0,       0,          1.0,               2       },
   { -0.5,       -0.866025,  0.5,               1       },
   { 0.5,        -0.866025,  1.0,               2       },
   { 1.0,        0,          0.5,               1       },
   { 0.5,        0.866025,   1.0,               2       },
   { -0.5,       0.866025,   0.5,               1       },
   { -1.0,       0,          1.0,               2       },

   { 0,          0,          1.41421,           2.82843 },
   { 0,          0,          0.707107,          1.41421 },
   { 0,          0,          1.41421,           2.82843 },
   { 0,          0,          0.707107,          1.41421 },
   { 0,          0,          1.41421,           2.82843 },
   { 0,          0,          0.707107,          1.41421 },
   { 0,          0,          1.41421,           2.82843 }
};

/* trimming curves data */
static float      trim_knot_vector1[7]  = {0,0,1,2,3,4,4};
static float     trim_knot_vector2[12] = {0,0,0,1,1,2,2,3,3,4,4,4};
```

```
/* boundaries of the surface parameters range */
static Xgl_pt_f2d trim_ctrl_arr1[5]     =
{
   {-1,-1}, {2,-1}, {2,1}, {-1,1}, {-1,-1}
};

/* control points for a circle centered at (0,0) with radius 0.5 */
static Xgl_pt_f2h trim_ctrl_arr2[9]     = {
   { 0.5,           0,           1        },
   { 0.353553,      -0.353553,   0.707107 },
   { 0,             -0.5,        1        },
   { -0.353553,     -0.353553,   0.707107 },
   { -0.5,          0,           1        },
   { -0.353553,     0.353553,    0.707107 },
   { 0,             0.5,         1        },
   { 0.353553,      0.353553,    0.707107 },
   { 0.5,           0,           1        }
};


void
nurbs_sphere (Xgl_object         ctx)
{

   Xgl_nurbs_surf                surface;
   Xgl_trim_loop_list            trimming;
   Xgl_nurbs_surf_simple_geom    hints;
   Xgl_trim_loop                 trim_loop[2];
   Xgl_trim_curve                trim_curves1, trim_curves2;

   /* Fill in the surface data structure */

   surface.order_u = 3;
   surface.order_v = 3;
   surface.num_knots_u = 10;
   surface.num_knots_v = 8;
   surface.knot_vector_u = knot_vector_u;
   surface.knot_vector_v = knot_vector_v;
   surface.ctrl_pts.pt_type = XGL_PT_F3H;
   surface.ctrl_pts.bbox = NULL;
   surface.ctrl_pts.num_pts = 35;
   surface.ctrl_pts.pts.f3h = ctrl_pts_arr;
```

```
/* Fill in the trimcurve data structure */
  trim_curves1.order = 2;
  trim_curves1.num_knots = 7;
  trim_curves1.knot_vector = trim_knot_vector1;
  trim_curves1.range.bmin = 0.0;
  trim_curves1.range.bmax = 4.0;
  trim_curves1.ctrl_pts.pt_type = XGL_PT_F2D;
  trim_curves1.ctrl_pts.bbox = NULL;
  trim_curves1.ctrl_pts.num_pts = 5;
  trim_curves1.ctrl_pts.pts.f2d = trim_ctrl_arr1;
  trim_curves1.trim_curve_vis = FALSE;
  trim_curves1.trim_curve_approx = XGL_TRIM_CURVE_VIEW_DEPENDENT;
  trim_curves1.trim_curve_approx_value = 0.5;

  trim_curves2.order = 3;
  trim_curves2.num_knots = 12;
  trim_curves2.knot_vector = trim_knot_vector2;
  trim_curves2.range.bmin = 0.0;
  trim_curves2.range.bmax = 4.0;
  trim_curves2.ctrl_pts.pt_type = XGL_PT_F2H;
  trim_curves2.ctrl_pts.bbox = NULL;
  trim_curves2.ctrl_pts.num_pts = 9;
  trim_curves2.ctrl_pts.pts.f2h = trim_ctrl_arr2;
  trim_curves2.trim_curve_vis = FALSE;
  trim_curves2.trim_curve_approx = XGL_TRIM_CURVE_VIEW_DEPENDENT;
  trim_curves2.trim_curve_approx_value = 0.5;

  trim_loop[0].num_curves = 1;
  trim_loop[0].curves = &trim_curves1;
  trim_loop[1].num_curves = 1;
  trim_loop[1].curves = &trim_curves2;

  trimming.num_loops = 2;
  trimming.trim_loops = trim_loop;

  /* Fill in the hints data structure */

  hints.surf_type = XGL_SURF_SPHERICAL;
  hints.geom_desc.spherical.center.x = 0.0;
  hints.geom_desc.spherical.center.y = 0.0;
  hints.geom_desc.spherical.center.z = 0.0;
  hints.geom_desc.spherical.radius = 0.5;
  hints.geom_desc.spherical.norm_flag = TRUE;

  /* Set various context attributes */
```

```
  xgl_object_set (ctx,
    XGL_CTX_VDC_MAP, XGL_VDC_MAP_ASPECT,
    XGL_CTX_NURBS_SURF_APPROX, XGL_SURF_RELATIVE_DC,
    XGL_CTX_NURBS_SURF_APPROX_VAL_U, 0.5,
    XGL_CTX_NURBS_SURF_PARAM_STYLE, XGL_SURF_ISO_CURVES,
    XGL_CTX_NURBS_SURF_ISO_CURVE_PLACEMENT,
    XGL_ISO_CURVE_BETWEEN_LIMITS,
    XGL_CTX_NURBS_SURF_ISO_CURVE_U_NUM, 15,
    XGL_CTX_NURBS_SURF_ISO_CURVE_V_NUM, 15,
    XGL_CTX_SURF_FRONT_FILL_STYLE, XGL_SURF_FILL_SOLID,
    XGL_3D_CTX_SURF_FACE_CULL, XGL_CULL_BACK,
    XGL_CTX_MARKER_SCALE_FACTOR, 6.0,
    XGL_CTX_MARKER_COLOR, &white_color,
    XGL_CTX_MARKER_DESCRIPTION, xgl_marker_circle,
    0);

   /* draw control points as markers */
   xgl_multimarker (ctx, &surface.ctrl_pts);

   /* Finally, draw the surface */
  xgl_nurbs_surface (ctx, &surface, &trimming, &hints, NULL, NULL);
}
```

Using Geometry Cache for Curve and Surface Rendering

The preceding sections of this chapter discussed rendering NURBS curves and surfaces directly. Improved performance for NURBS curve and surface rendering can be achieved using a Geometry Cache object. As described in Chapter 16, “Caching Geometry”, a Gcache object is used to store an XGL primitive and break it down into a simpler form for fast, per-frame display.

The application has the choice of creating a Gcache that stores a tessellation-independent form of the curve or surface, which is dynamically tessellated into lines or triangles each time the curve or surface is displayed, a tessellated form of the curve or surface, or a combination of forms in which tessellation is regenerated only when necessary. The application can choose the appropriate mode depending on memory availability, speed and display quality requirements, or the complexity of NURBS data and viewing requirements. For more information on NURBS Gcache objects, refer to Chapter 16, “Caching Geometry” or the *XGL Reference Manual*.

Getting the Best Out of XGL NURBS

The following tips will enable an application to get the best performance using the XGL NURBS implementation.

- Performance with NURBS curves and surfaces may be significantly improved using Gcache objects. If the application creates a NURBS Gcache once, and then calls `xgl_display_gcache()` for every frame, performance will increase by an order of magnitude because substantial preprocessing is performed when the Gcache is created. See Chapter 16, "Caching Geometry" for information on Gcache objects and Gcache NURBS operators and attributes.
- As mentioned on page 253, if the application supplies hints about simple surface geometry, a major performance improvement can be expected when there is lighting and shading.
- Enabling clipping will instruct XGL to discard portions of the NURBS outside of the clipping region. This will considerably speed up the display.
- Using front/back face culling will speed up display substantially.
- If the device does not have fast triangle shading, wireframe mode can be simulated by turning off lighting, turning on edge display, and specifying isoparametric lines. Further, if the surface interior fill style is `XGL_SURF_FILL_SOLID`, a hidden-line image is generated.
- Using static approximation types (as opposed to dynamic approximation types) may not yield any performance gain. In fact, in cases where there are substantial view changes and zooms, a fixed approximation value does not make much sense. The application is encouraged to use dynamic approximation types, especially the `RELATIVE` types. By taking advantage of this simple rendering quality measure, the application can trade-off speed for display quality.

Further Reading

There are numerous books on NURBS geometry and mathematics. For more information on the characteristics of NURBS curves and surfaces, refer to the following:

- Farin, Gerald. *Curves and Surfaces for Computer Aided Geometric Design*, Second Edition, Academic Press, Inc., San Diego, CA, 1990.
- Bartels, R., et al. *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*, Morgan Kaufmann Publishers, Los Altos, California, 1987.

Transforms 9

This chapter discusses the XGL Transform object and its operators. It includes information on the following topics:

- Concepts of Transform objects
- Transform operators:
  - Interface-level representations
  - Basic utilities
  - Concatenation
- Examples using Transform operators

Introduction to the Transform Object

A transformation is a linear mapping of geometric data from one coordinate space to another. XGL stores the information for this mapping in a Transform object and applies the mapping to the *x*, *y*, and *z* coordinates of geometric primitives. Transforms are the basic building blocks for sequentially manipulating geometry. They ***model*** a complicated scene from simple geometric primitives, ***view*** the scene with a particular orientation and perspective, and place the image in a window. These operations are associated with the view model, which is discussed in Chapter 10, "View Model."

At the interface level, application programmers can think of Transforms as having matrix representations. XGL treats position vectors as homogeneous row vectors, which normally have a homogeneous component w set to 1. In the

2D case, the matrix is 3×3. This matrix can represent any sequence of translations, rotations, and scalings from one space to another in the following way:

$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

This representation extends to 3D, for which the matrix is 4×4.

Transforms can operate in 2D or 3D space. 2D Transforms can be represented in integer, floating point, or double precision format; 3D Transforms can be represented in floating point or double precision format. The dimension and data type of a Transform are attributes of the Transform object.

Creating a Transform Object

A new Transform object is created with the `xgl_object_create()` operator using a parameter `type` of `XGL_TRANS` and the parameter `desc` of `NULL`. The `xgl_object_create()` operator initializes the new Transform to identity and sets the initial values of the specified attributes.

Transform Operators

The XGL library provides operators for manipulating Transforms, including rotation, translation, and scaling operators. These operators can concatenate a rotation, translation, or scale transformation with an existing transformation. XGL also supplies other utility operators to compute the values of transformations.

Note – The Transforms used as parameters in the Transform operators must be created with `xgl_object_create()` before being used by the Transform operators. When a Transform is no longer needed by an application, it should be destroyed using `xgl_object_destroy()` to free the resources associated with it.

Rotation Operator

The matrices for rotation by angle θ in the right-handed sense have the following form. Note that 2D rotations are counterclockwise when the x-axis points right and the y-axis points up. They are clockwise when the x-axis points right and the y-axis points down.

For 2D: $\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$

For 3D about the z-axis: $\begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

For 3D about the y-axis: $\begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

For 3D about the x-axis: $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

The Rotation operator replaces a Transform with a Rotation Transform constructed from the given parameters. It can also preconcatenate or postconcatenate a Rotation Transform with the given Transform. When a matrix is postconcatenated, the matrix is multiplied on the right of the original Transform matrix because XGL represents points as row vectors. The Rotation operator is:

```
void xgl_transform_rotate (
    Xgl_trans           trans,
    double              angle,
    Xgl_axis            axis,
    Xgl_trans_update    update);
```

Scaling Operator

The matrices for scaling have the following form:

For 2D: $\begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$

For 3D: $\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

The Scale operator replaces a Transform with a Scale Transform constructed from the given parameters. It can also preconcatenate or postconcatenate a Scale Transform with the given Transform. Scale factors set to zero may result in errors. The Scale operator is defined as:

```
void xgl_transform_scale(
    Xgl_trans           trans,
    Xgl_pt              *scale_factors,
    Xgl_trans_update    update);
```

Translation Operator

The matrices for translation have the following form:

For 2D: $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 1 \end{bmatrix}$

For 3D: $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$

The Translation operator replaces an existing Transform with a Translation Transform constructed from the given parameters. It can also preconcatenate or postconcatenate a Translation Transform to the given Transform. The Translation operator is defined as:

```
void xgl_transform_translate(
    Xgl_trans           trans,
    Xgl_pt              *offset,
    Xgl_trans_update    update);
```

Reading and Writing Transforms

Applications may need to read or write the data in a Transform if the formulation of the transformation is more complicated than a sequence of translations, rotations, and scalings. XGL provides operators for reading a Transform into an array and writing an array into a Transform. For 3D, the arrays are 4×4. For 2D, the arrays are 3×2 because the last column is always $[0,0,1]^T$.

The `xgl_transform_read()` operator reads a Transform's matrix into an array supplied by the application:

```
void_xgl_transform_read (
    Xgl_trans           src_trans,
    void                *matrix);
```

The `xgl_transform_write_specific()` operator writes an array into a Transform object and specifies the membership of the matrix to the matrix groups:

```
void xgl_transform_write_specific (
    Xgl_trans           dest_trans,
    void                *matrix,
    Xgl_trans_member    membership);
```

A matrix group is a particular type of matrix with a well-defined set of properties. An application can give XGL hints about the matrix it is writing into a Transform by specifying the matrix's membership in a matrix group.

Membership in a matrix group can significantly improve the performance of most operations involving transformations. The matrix groups are listed in Table 9-1. For the form of each matrix, see the `xgl_transform_write_specific()` manual page in the *XGL Reference Manual*.

Table 9-1 Transform Memberships in Matrix Groups

Value	Description
XGL_TRANS_MEMBER_IDENTITY	Identity matrix.
XGL_TRANS_MEMBER_TRANSLATION	Translation matrix.
XGL_TRANS_MEMBER_SCALE	Scale matrix.
XGL_TRANS_MEMBER_ROTATION	Rotation matrix.
XGL_TRANS_MEMBER_WINDOW	A window matrix is a combined scale and translation matrix.
XGL_TRANS_MEMBER_SHEAR_SCALE	Shear-scale matrix. This matrix type is 3D only.
XGL_TRANS_MEMBER_LENGTH_PRESERV	A length-preserving matrix can translate, rotate, and reflect about an axis. These operations preserve the distance between any two points and the angle between any two intersecting lines.
XGL_TRANS_MEMBER_ANGLE_PRESERV	An angle-preserving matrix can translate, rotate, and reflect about an axis, and scale by an amount which is uniform in all directions. These operations preserve the angle between any two intersecting lines.
XGL_TRANS_MEMBER_AFFINE	An affine matrix can translate, rotate, reflect, scale anisotropically, and shear. Lines remain straight, and parallel lines remain parallel, but the angle between intersecting lines can change.
XGL_TRANS_MEMBER_LIM_PERSPECTIVE	Limited perspective matrix. This matrix type is 3D only.

The `xgl_transform_write()` operator writes an array supplied by the application into the Transform's matrix.

```
void xgl_transform_write (
    Xgl_trans          dest_trans,
    void               *matrix);
```

Other Transform Operators

The following transformation operators provide utility functions. For operators that require a source Transform and a destination Transform, the two Transforms can be the same.

Copy

The Copy operator copies the contents of the matrix associated with the Source Transform to the matrix associated with the Destination Transform. 2D and 3D Transforms can be mixed in this operation.

```
void xgl_transform_copy (
    Xgl_trans          dest_trans,
    Xgl_trans          src_trans);
```

Identity

The Identity operator sets the Transform's matrix to the Identity. An Identity Transform has no effect on geometric data passing through it. The Identity operator is defined as:

```
void xgl_transform_identity (Xgl_trans trans);
```

Invert

The Invert operator inverts a Transform's matrix. The inverse of a matrix, when multiplied with the original matrix, produces the identity matrix. A Transform may be singular, having no unique inverse. The operator normally returns *dest_trans*, but will return NULL if the Transform is singular. The numerical accuracy depends on the Transform's data type. The Invert operator is:

```
Xgl_trans xgl_transform_invert(
    Xgl_trans          dest_trans,
    Xgl_trans          src_trans);
```

Note – Do not mix 2D and 3D Transforms with the `xgl_transform_invert()` operator.

Multiply

The Multiply operator multiplies two Transforms. All Transforms passed to this operator must have the same dimension. The Multiply operator is defined as:

```
void xgl_transform_multiply (
    Xgl_trans          dest_tr,
    Xgl_trans          left_src_tr,
    Xgl_trans          right_src_tr);
```

Note – Do not mix 2D and 3D Transforms with the `xgl_transform_multiply()` operator.

Transform Point

The Transform Point operator transforms a single point by treating the point as a row vector and multiplying it by the Transform's matrix. The result is stored back in the memory allocated for the point, overwriting the original point.

```
void xgl_transform_point (
    Xgl_trans           trans,
    Xgl_pt              *point);
```

Transform Point List

The Transform Point List operator transforms a list of points, treating each point as a row vector and multiplying it by the Transform's matrix. This operator is defined as:

```
void xgl_transform_point_list(
    Xgl_trans           trans,
    Xgl_pt_list         *src_pt_list,
    void                *dest_pts);
```

Transpose

The Transpose operator transposes a Transform's matrix. The *i*th column of a transposed matrix is the *i*th row of the original matrix. The *i*th row of a transposed matrix is the *i*th column of the original matrix.

This operator may be useful if an application uses `xgl_transform_write()` and treats position vectors as column vectors instead of row vectors like XGL. Another application of this operator is with rotation Transforms: the inverse is its transpose. 2D and 3D Transforms can be mixed using this operator. The Transpose operator is:

```
void xgl_transform_transpose (
    Xgl_trans           dest_trans,
    Xgl_trans           src_trans);
```

Transform Attributes

The following Transform attributes configure Transforms.

XGL_TRANS_DATA_TYPE
This attribute defines the data type of the matrix associated with a Transform. For 2D Transforms, the data type can be integer, floating point, or double precision. For 3D Transforms, the data type can be floating point or double precision.

XGL_TRANS_DIMENSION
This attribute defines whether the space in which the Transform operates is 2D or 3D. The attribute can be set to XGL_TRANS_2D, in which case the Transform operates in two-dimensional space, or XGL_TRANS_3D, so that the Transform operates in three-dimensional space.

Examples Using the Transform Object

This section explains the example 2D and 3D XGL programs at the end of this chapter. The first example shows an untransformed triangle in 2D space. In the next three examples, the triangle is translated, rotated, and scaled. The final example shows a method for creating a View Transform to give an oblique parallel projection of a cube.

The transformation example programs are to be used within the framework of the general utility program ex_utils.c. Each example is a fragment of a larger program. The complete program includes ex_utils.c and tran_main.c, both of which are listed in Appendix B, as well as all the examples listed in this section. To compile the complete program, type make tran in the example program directory. The compiled program allows you to look at all the transformation example programs.

2D Transform Examples

The example program ex_utils.c creates a 2D Context. This Context has a pipeline consisting of Transforms and structures for describing clipping and mapping. The pipeline implements the XGL view model. Creation of a 2D Context automatically causes creation of all the Transforms in the Context's view model. Therefore, the example programs need not explicitly create the Context's View Transform.

The 2D example programs are alike, except in the particular Transform operator each invokes. The first example, which draws the triangle in its original position, obtains a handle to the 2D Context's View Transform. When the handle, `view_trans`, is the target of a Transform operator, the Context becomes aware of the change to its pipeline. In the first example, `view_trans` is a parameter of `xgl_transform_identity()`, and sets the View Transform to the Identity matrix.

The example program set the orientation of Virtual Device Coordinates (VDC) so the x-axis points right and the y-axis points up. (Chapter 10, "View Model," describes this in more detail.) This places the origin in the lower left corner of the window. Since the View Transform is the Identity, the vertices defining the triangle are measured in pixels.

The second, third, and fourth examples translate, rotate, and scale the triangle, respectively. The translation and scaling operators take a vector that specifies the appropriate information. The translation example translates the triangle by 100 in the x-direction and 10 in the y direction. The scale example scales the triangle by 2 in both the *x*- and y directions.

In the rotation example, the rotation operator takes an angle in radians. A positive angle results in a counterclockwise rotation here because the y-axis points up in this example. XGL ignores the third parameter for 2D rotations. In 3D, a positive angle gives a right-handed rotation around the axis specified by the third parameter.

The translation, rotation, and scaling operators all take a parameter that specifies how to combine the requested transformation with the target Transform. The second, third, and fourth examples all replace the contents of the target Transform to perform a pure translation, rotation, and scaling, respectively. An application can also concatenate a requested transformation with the target Transform. Preconcatenation and postconcatenation are both supported. Concatenation of two Transforms produces a Transform that gives the same result as when applying the two original Transforms in sequence.

Displaying the Original Geometry

This code fragment displays the triangle before the application of any Transforms. Figure 9-1 shows the result.

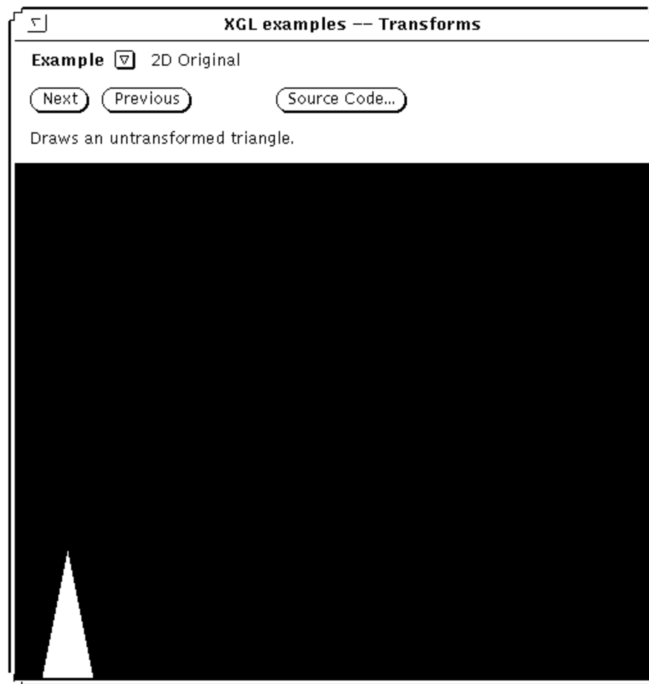


Figure 9-1 Output of `tran_2d_orig.c`

Code Example 9-1 Geometry Before Transformations

```
/*
 * tran_2d_orig.c
 */

#include <xview/xview.h>
#include <xgl/xgl.h>

#include "ex.h"

extern Xgl_pt_list   pl_2d;
```

```
void
tran_2d_orig (Xgl_object   ctx)
{
    Xgl_color            color;
    Xgl_object           view_trans;

    color = cyan_color;

    xgl_object_set (ctx,
                  XGL_CTX_VDC_ORIENTATION, XGL_Y_UP_Z_TOWARD,
                  XGL_CTX_SURF_FRONT_COLOR, &color,
                  0);

    /* Get the view transform and set it to the identity */
    xgl_object_get (ctx, XGL_CTX_VIEW_TRANS, &view_trans);
    xgl_transform_identity (view_trans);

    /* Draw a 2D polygon */
    xgl_polygon (ctx, XGL_FACET_NONE, NULL, NULL, 1, &pl_2d);
}
```

Translating the Geometry

This function illustrates how to set up a Transform object for translation. Figure 9-2 illustrates the output.

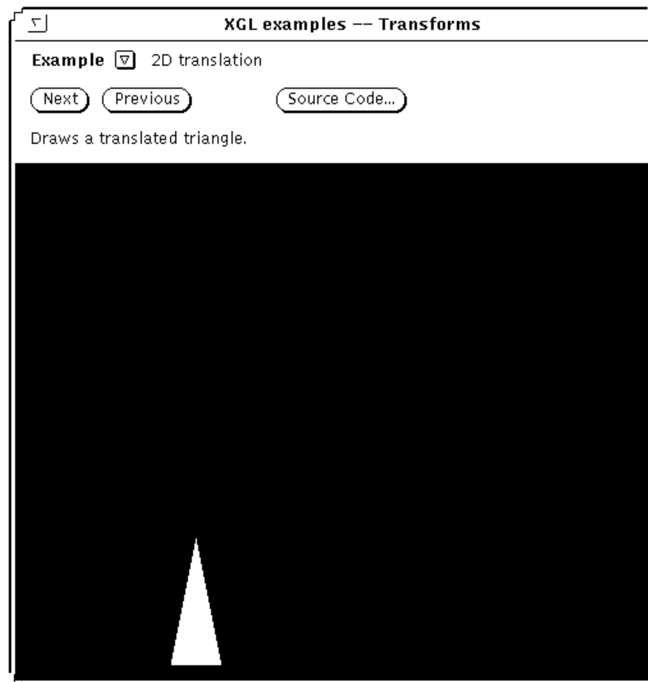


Figure 9-2 Output of `tran_2d_transl.c`

Code Example 9-2 Translation Example

```
/*
 * tran_2d_transl.c
 */

#include <xview/xview.h>
#include <xgl/xgl.h>

#include "ex.h"

extern Xgl_pt_list    pl_2d;
```

```
void
tran_2d_translate (Xgl_object    ctx)
{
    Xgl_color            color;
    Xgl_object           view_trans;
    Xgl_pt               pt;
    Xgl_pt_f2d           pt_f2d;

    color = cyan_color;

    xgl_object_set (ctx,
                  XGL_CTX_VDC_ORIENTATION, XGL_Y_UP_Z_TOWARD,
                  XGL_CTX_SURF_FRONT_COLOR, &color,
                  0);

    xgl_object_get (ctx, XGL_CTX_VIEW_TRANS, &view_trans);

    pt.pt_type = XGL_PT_F2D;
    pt.pt.f2d = &pt_f2d;
    pt_f2d.x = 100.0;
    pt_f2d.y = 10.0;

    xgl_transform_translate (view_trans, &pt, XGL_TRANS_REPLACE);

    xgl_polygon (ctx, XGL_FACET_NONE, NULL, NULL, 1, &pl_2d);
}
```

Rotating the Geometry

This code fragment illustrates how to set up a Transform object for rotation. Figure 9-3 shows the rotated object.

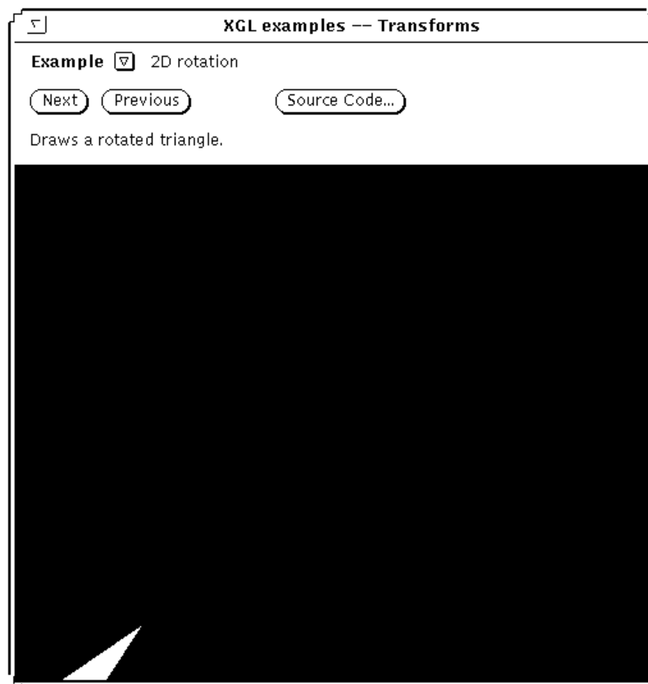


Figure 9-3 Output of `tran_2d_rot.c`

Code Example 9-3 Rotation Example

```
/*
 * tran_2d_rot.c
 */

#include <xview/xview.h>
#include <xgl/xgl.h>
#include "ex.h"

#define    PI      3.141592654
```

```
extern Xgl_pt_list   pl_2d;

void
tran_2d_rotate (Xgl_object   ctx)
{
    Xgl_color            color;
    Xgl_object           view_trans;

    color = cyan_color;

    xgl_object_set (ctx,
                  XGL_CTX_VDC_ORIENTATION, XGL_Y_UP_Z_TOWARD,
                  XGL_CTX_SURF_FRONT_COLOR, &color,
                  0);

    /* Get the view transform and set it to a rotation transform */
    xgl_object_get (ctx, XGL_CTX_VIEW_TRANS, &view_trans);
    xgl_transform_rotate (view_trans, -PI / 4.0, XGL_AXIS_Z,
                   XGL_TRANS_REPLACE);

    /* Draw a 2D polygon */
    xgl_polygon (ctx, XGL_FACET_NONE, NULL, NULL, 1, &pl_2d);
}
```

Scaling the Geometry

This example illustrates how to set up the Transform object for scaling. Figure 9-4 shows the scaled object.

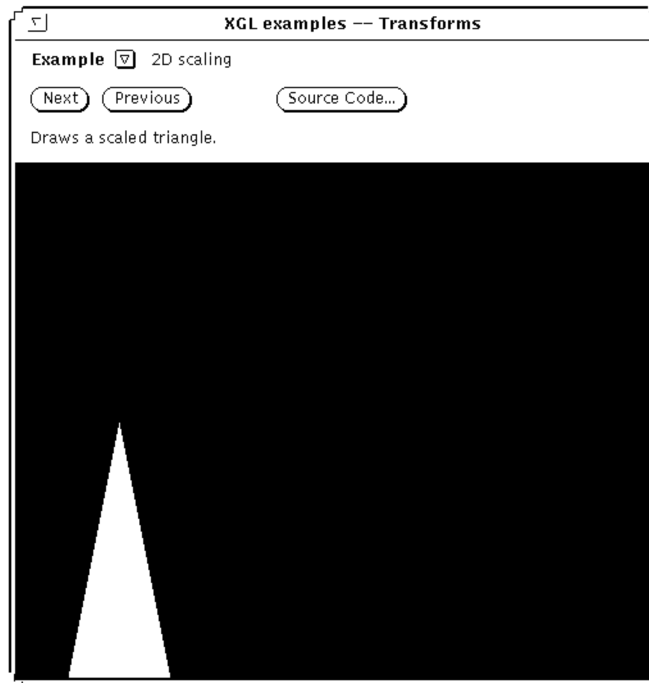


Figure 9-4 Output of `tran_2d_scale.c`

Code Example 9-4 Scale Example

```
/*
 * tran_2d_scale.c
 */

#include <xview/xview.h>
#include <xgl/xgl.h>
#include "ex.h"

extern Xgl_pt_list   pl_2d;
```

```
void
tran_2d_scale (Xgl_object     ctx)
{
    Xgl_color            color;
    Xgl_object           view_trans;
    Xgl_pt               pt;
    Xgl_pt_f2d           pt_f2d;

    color = cyan_color;

    xgl_object_set (ctx,
                  XGL_CTX_VDC_ORIENTATION, XGL_Y_UP_Z_TOWARD,
                  XGL_CTX_SURF_FRONT_COLOR, &color,
                  0);

    /* Get the view transform and set it to a scale transform */
    xgl_object_get (ctx, XGL_CTX_VIEW_TRANS, &view_trans);

    pt.pt_type = XGL_PT_F2D;
    pt.pt.f2d = &pt_f2d;
    pt_f2d.x = 2.0;
    pt_f2d.y = 2.0;

    xgl_transform_scale (view_trans, &pt, XGL_TRANS_REPLACE);

    /* Draw a 2D polygon */
    xgl_polygon (ctx, XGL_FACET_NONE, NULL, NULL, 1, &pl_2d);
}
```

3D Transform Example

The final example, `tran_3d.c`, illustrates a method for constructing a 3D View Transform. It begins by creating a temporary Transform to hold intermediate results. By default, a new Transform holds floating-point values and handles 3D points. An application can change the dimension or data type, or both, of a Transform at creation or at any time afterward. However, the Transforms associated with a 3D Context must be 3D; those associated with a 2D Context must be 2D.

The example program applies a sequence of one translation and three rotations, with the operators appearing in the 2D example programs. The first operator replaces the contents of the Transform with a translation, and the succeeding rotations follow the effect of the translation. Postconcatenation of each rotation to the preceding sequence of transformations successively builds a single Transform that gives the cumulative effect.

Having completed the basic operations for building the desired view transformation, the example gets a handle to the 3D Context's View Transform and copies the result of the preceding calculations to this Transform in the Context's view model. The example then sets some additional view model attributes (described in Chapter 10, "View Model") and draws a projection of a wireframe cube using the view transformation just calculated. The final step is destruction of the Transform created at the beginning for holding intermediate results. Figure 9-5 on page 287 shows the wireframe cube.

An explanation of the mathematical principles used in these examples can be found in *Principles of Interactive Computer Graphics*, pages 54-55 and 348-351[1]. The XGL convention for positive angles corresponding to rotations in the right-handed sense, however, is opposite to the convention proposed by Newman and Sproull.

---

1. Newman, William M. and Robert F. Sproull, *Principles of Interactive Computer Graphics*, McGraw-Hill, New York, N.Y., 2nd edition, 1979.

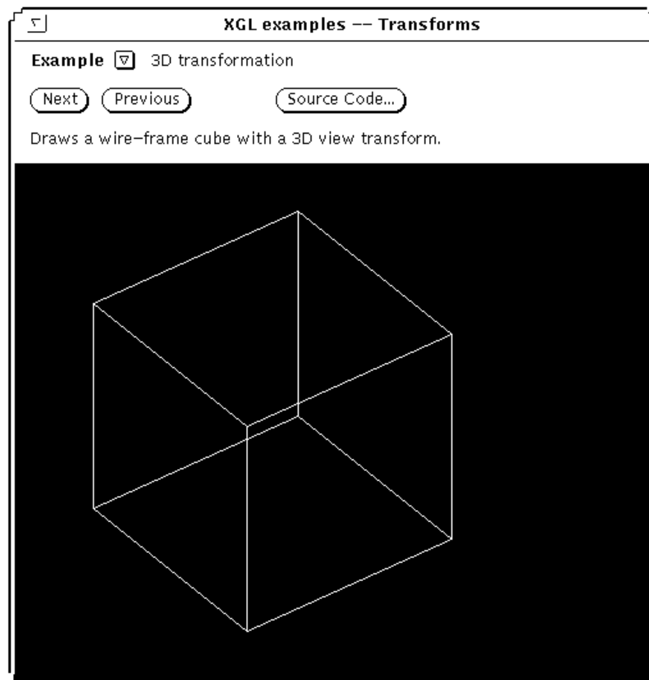


Figure 9-5 Output of `tran_3d.c`

Code Example 9-5 3D Transform Example

```
/*
 * tran_3d.c
 */
#include <math.h>
#include <xview/xview.h>
#include <xgl/xgl.h>

#include "ex.h"

#define    PI  3.141592654

extern Xgl_object        sys_st;
extern Xgl_pt_list       pl_3d[];
```

```
void
tran_3d (Xgl_object      ctx)
{
    Xgl_color            color;
    Xgl_object           trans;
    Xgl_object           view_trans;
    Xgl_pt               pt;
    Xgl_pt_f3d           pt_f3d;
    Xgl_bounds_d3d       vdc_window;

    /* Create a 3D floating-point transform */
    trans = xgl_object_create(sys_st, XGL_TRANS, NULL, NULL);

    /* Set up view transform */

    /* Translate the origin of VDC to (6.0, 8.0, 7.5) in WC */
    pt.pt_type = XGL_PT_F3D;
    pt.pt.f3d = &pt_f3d;
    pt_f3d.x = -6.0;
    pt_f3d.y = -8.0;
    pt_f3d.z = -7.5;
    xgl_transform_translate (trans, &pt, XGL_TRANS_REPLACE);

    /* Swing the y-axis of VDC so that it is vertical in WC */
    xgl_transform_rotate (trans, -PI / 2, XGL_AXIS_X,
                          XGL_TRANS_POSTCONCAT);

    /* Swivel the z-axis of VDC away from WC's z-axis */
    xgl_transform_rotate (trans, atan (pt_f3d.x / pt_f3d.y) + PI,
                          XGL_AXIS_Y, XGL_TRANS_POSTCONCAT);

    /* Tip VDC so that its z-axis points away from WC's origin */
    xgl_transform_rotate (trans, atan (-pt_f3d.z /
                             sqrt (pt_f3d.x * pt_f3d.x +
                                   pt_f3d.y * pt_f3d.y)),
                       XGL_AXIS_X, XGL_TRANS_POSTCONCAT);

    /* Set context's view transform */
    xgl_object_get (ctx, XGL_CTX_VIEW_TRANS, &view_trans);
    xgl_transform_copy (view_trans, trans);

    /* Set line color and VDC parameters */
    color = yellow_color;
    vdc_window.xmin = -2.0;
    vdc_window.xmax = 2.0;
    vdc_window.ymin = -2.0;
```

```
    vdc_window.ymax = 2.0;
    vdc_window.zmin = -14.5;
    vdc_window.zmax = -10.5;
    xgl_object_set (ctx, XGL_CTX_LINE_COLOR, &color,
                  XGL_CTX_VDC_ORIENTATION, XGL_Y_UP_Z_TOWARD,
                  XGL_CTX_VDC_MAP, XGL_VDC_MAP_ASPECT,
                  XGL_CTX_VDC_WINDOW, &vdc_window,
                  0);

    /* Draw wireframe representation of data */
    xgl_multipolyline (ctx, NULL, 4, pl_3d);

    /* Clean up */
    xgl_object_destroy (trans);
}
```

View Model 10

This chapter discusses the XGL viewing pipeline. It includes information on the following topics:

- Coordinate systems and clipping
- 2D and 3D Context view model attributes
- 2D and 3D viewing and clipping pipelines

Introduction to the XGL Viewing Pipeline

The *viewing pipeline* is a series of transformations and clip operations applied to a graphics primitive before rendering. It transforms geometric coordinate values into a range suitable for display on the output device. The XGL viewing pipeline moves the geometric data from an application's space to rendering space via a set of transformation and clipping attributes. XGL has distinct and separate viewing pipelines for 2D and 3D. The XGL 2D viewing model is an affine[1] model. The XGL 3D viewing model is a homogeneous[2] model.

1. *Affine* means a geometric mapping that preserves the straightness of lines and the parallelism of parallel lines but possibly alters the distances between points or the angles between lines.

2. A *homogeneous* representation is a mapping of a point in n-dimensional ordinary coordinates to $(n+1)$-dimensional homogeneous coordinates. Given a point in three-dimensional ordinary coordinates $[x, y, z]$, the homogeneous representation is $[wx, wy, wz, w]$ for any $w \neq 0$. The inverse mapping from homogeneous coordinates to ordinary coordinates — via division by w — is called *projection*.

The XGL viewing model consists of four coordinate systems: Model Coordinates (MC), World Coordinates (WC), Virtual Device Coordinates (VDC), and Device Coordinates (DC). The Transforms presented in Chapter 9, "Transforms", map the geometry between any two sequential coordinate systems in the pipeline. Each Context has its own set of Transforms to implement these mappings. All Transforms default to the *Identity Transform* (which makes no change to the coordinate values); applications can ignore the Transforms, unless needed.

Clipping is performed in two places in the 2D pipeline and three places in the 3D pipeline. In both the 2D and 3D pipeline, view clipping can take place in VDC coordinates, and clipping to DC limits is always applied to ensure that the drawing fits into the display area (either a window or a non-raster device). In the 3D pipeline, model clipping can also be defined in Model Coordinates.

Coordinate Systems

Coordinate systems define the geometric environment in which the graphic primitive coordinate values are defined. Coordinate systems associated with the 2D pipeline are two-dimensional. Coordinate systems associated with the 3D pipeline are three-dimensional. An application can pass only 2D points to a 2D pipeline and only 3D points to a 3D pipeline. The viewing pipeline consists of the following coordinate systems:

Model Coordinates (Local and Global MCs)
Model Coordinates give the frame of reference in which a geometric model is defined. For example, a model of a bolt might be defined in inches or millimeters, whereas a ship might be defined in feet or meters. The application controls the placement of each Model Coordinate System into the next coordinate system of the pipeline, World Coordinates, with a transformation called the Model Transform.

World Coordinates (WC)
The World Coordinate System is common to all geometry for a particular view. XGL performs lighting in this space when 3D applications require illumination. The individual objects in the image to be rendered are represented here in proper perspective relative to each other. Viewing, or the application of a View Transform, maps the scene into a device-independent Virtual Device Coordinate System.

Virtual Device Coordinates (VDC)

XGL defines a Virtual Device Coordinate System to isolate applications from physical Device Coordinates, which might be different for different devices such as raster devices or file output devices. XGL performs view clipping in this space when applications require it.

Device Coordinates (DC)

Device Coordinates are the actual coordinates of the Device. If the Device is a Raster, the units of Device Coordinates are in pixels. In Device Coordinates, (0,0,0) is the upper left corner, where the axes are oriented right for positive *x*, down for positive *y*, and away for positive *z*.

The Context object includes information that specifies the mapping and clipping of geometric data between coordinate spaces. The Local Model Transform maps simple geometric primitives comprising a piece of the object to a global modeling space where the object is assembled. The Global Model Transform maps objects into the final scene in world coordinate space. The View Transform determines the application's viewing direction, orientation, and perspective.

Taken together, the Local Model Transform and the Global Model Transform map coordinate values from Model Coordinates to World Coordinates. The View Transform maps coordinate values from World Coordinates to Virtual Device Coordinates. The mapping from Virtual Device Coordinates to Device Coordinates is automatic and is computed by comparing the VDC and DC ranges (this mapping is limited to scaling and translation). It is also possible to map coordinates values directly from Model Coordinates to Device Coordinates using the MC-to-DC Transform. The MC-to-DC Transform results from the concatenation of the Model Transform, the View Transform, the VDC Transform, and the Device's DC orientation.

Figure 10-1 on page 294 and Figure 10-2 on page 295 illustrate the 2D and 3D view models, showing the transformations and the coordinate systems in the XGL viewing pipeline.

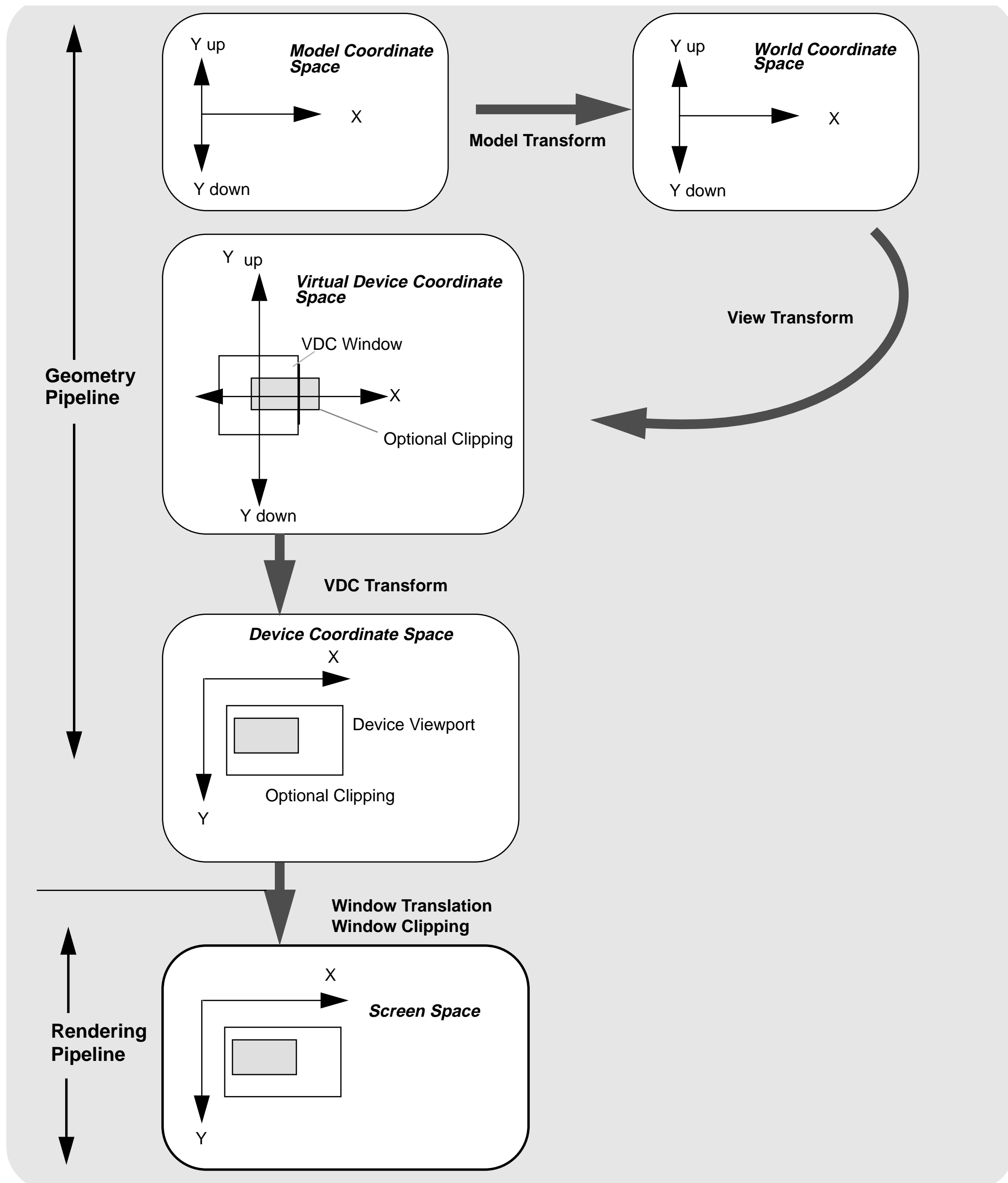


Figure 10-1 2D View Model

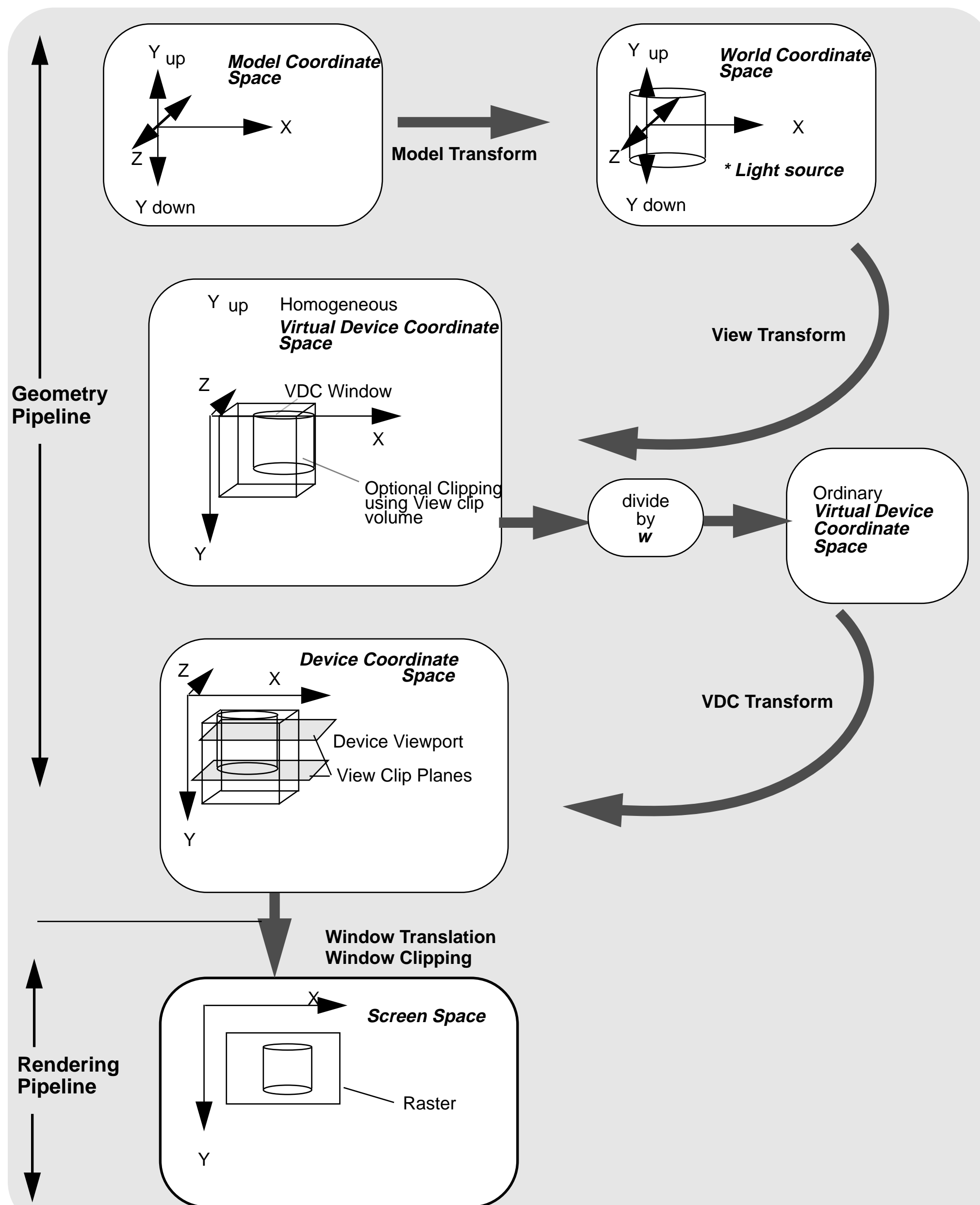


Figure 10-2 3D View Model

How to Use Transforms with the View Model

When an application creates a 2D or 3D Context object, the Context object creates a default transformation pipeline, which includes a Local Model Transform, a Global Model Transform, a Model Transform, an MC-to-DC Transform, and a View Transform. As part of the object initialization, the Context object sets these Transform objects to Identity. If a 3D Context is created, a Normal Transform is also created. An application can get the handles of these Transforms from the Context object. Although the Model Transform, the MC-to-DC Transform, and the Normal Transform cannot be modified, the application can change the values of the Local Model Transform, the Global Model Transform, and the View Transform using the Transform operators. Resetting the View Transform is shown in the code fragment below.

```
Xgl_trans      view_trans;

/* Get the view transform and set it to a rotation transform */
xgl_object_get (ctx, XGL_CTX_VIEW_TRANS, &view_trans);
xgl_transform_rotate (view_trans, -PI / 4.0, XGL_AXIS_Z,
                      XGL_TRANS_REPLACE);
```

To get the handles to the Context's Transform objects, use these attributes:

Table 10-1 Getting Context Transform Handles

To get a handle to this Transform:	Use this attribute:
Local Model Transform	XGL_CTX_LOCAL_MODEL_TRANS
Global Model Transform	XGL_CTX_GLOBAL_MODEL_TRANS
Model Transform	XGL_CTX_MODEL_TRANS (RO)
View Transform	XGL_CTX_VIEW_TRANS
MC-to-DC Transform	XGL_CTX_MC_TO_DC_TRANS (RO)
Normal Transform (3D only)	XGL_3D_CTX_NORMAL_TRANS (RO)

When the application gets the handle to a Transform object, both the application and the Context, which owns the Transform objects, have a handle to the Transform in the Context's pipeline. If this handle is the destination parameter of a Transform operator, the Context automatically knows about the change to one of its Transforms. Note that if application-created Transforms are

used as parameters by the Transform operators, the Transforms must be created before they can be used by the Transform operators. When a Transform is no longer needed in an application, it should be destroyed using the `xgl_object_destroy()` operator to free its resources.

Alternatively, an application can create a new Transform object for the Local Model Transform, the Global Model Transform, or the View Transform and set the Context attribute to the new Transform. However, the recommended way of modifying a transformation is to retrieve the relevant transformation matrix from the Context object and modify it rather than creating a new Transform object.

Note – If you set the `XGL_CTX_LOCAL_MODEL_TRANS`, `XGL_CTX_GLOBAL_MODEL_TRANS`, or `XGL_CTX_VIEW_TRANS` attributes to a Transform whose handle is retrieved from the same Context or a different Context, the original attribute and the current attribute use the same Transform. Changing the Transform of one Context attribute has the same effect on the other Context attribute. Be sure that you are aware of the consequences when you share a Transform in this manner. Do not attempt to set these three attributes to a handle of a Transform retrieved from `XGL_CTX_MODEL_TRANS`, `XGL_CTX_MC_TO_DC_TRANS`, or `XGL_3D_CTX_NORMAL_TRANS`, because these Transforms are read-only.

Using XGL 2D and 3D Viewing

This section provides information on using the XGL viewing pipeline.

Model Coordinate Space and Model Transforms

An application passes graphical primitives to XGL in Model Coordinates (MC). The application can define any number of Model Coordinate Systems. Their relation to each other is determined by their locations and orientations with respect to World Coordinates (WC). The application controls the placement of each Model Coordinate System in World Coordinates by a transformation called the Model Transform.

Applications using the 2D or 3D pipeline can specify the Model Transform in two levels with a Local Model Transform and a Global Model Transform. The Model Transform is the Local Model Transform concatenated with the

succeeding Global Model Transform. The Model Transform is affected by changes to either the Local Model Transform or the Global Model Transform. When two Transforms are concatenated to form a single Transform, the single Transform behaves as both Transforms applied in succession. These Transforms are given by the XGL_CTX_LOCAL_MODEL_TRANS, XGL_CTX_GLOBAL_MODEL_TRANS, and XGL_CTX_MODEL_TRANS attributes.

Applications written for 3D Contexts also have a Normal Transform, which maps normal vectors from Model Coordinates to World Coordinates. The Normal Transform is the inverse of the Model Transform.

XGL_CTX_MODEL_TRANS and XGL_3D_CTX_NORMAL_TRANS are read-only attributes for the Transform objects that map position or normal vectors from Model Coordinates to World Coordinates. Both can be changed only by setting the Local or Global Model Transforms.

Note – Evaluation of the Model Transform and the Normal Transform is deferred until either XGL needs the Transform internally or the application explicitly gets it. Before attempting to use the Model Transform or the Normal Transform, call xgl_object_get() to get XGL_CTX_MODEL_TRANS or XGL_3D_CTX_NORMAL_TRANS to ensure that XGL evaluates the Transform if it requires updating.

The default value for the Model Transforms and the Normal Transform is the Identity Transform.

World Coordinate Space and the View Transform

Viewing projects a scene in World Coordinates onto an image plane. This step is similar to selecting a camera location, a direction and orientation for holding the camera, and the focal length of the lens, which determines the field of view and perspective. The transformation pipeline accomplishes this by passing the geometrical data assembled into World Coordinates through the View Transform into the Virtual Device Coordinate Space. The XGL_CTX_VIEW_TRANS attribute is the View Transform object. When the application defines the View Transform, it must take into account both the view clipping bounds defined with the attribute XGL_CTX_VIEW_CLIP_BOUNDS and the orientation of VDC space as defined by XGL_CTX_VDC_ORIENTATION. The View Transform, view clip bounds, and VDC orientation collectively determine the generated image in VDC space.

For 3D, the View Transform may include a *perspective transformation*, which would change an implicit *w* component of the geometric data from 1.0 to some other value. XGL allows in VDC both homogeneous coordinates (*x, y, z, w*) and ordinary coordinates (*x, y, z*). The geometry is projected into ordinary 3D space after the view clipping (specified by the XGL_CTX_CLIP_PLANES and XGL_CTX_VIEW_CLIP_BOUNDS attributes).

The default value for the View Transform is the Identity Transform.

Virtual Device Coordinate Space and the VDC Transform

Virtual Device Coordinate space isolates viewing from physical Device Coordinates. Each Device may have its own units, so the VDC-to-DC mapping allows much of the pipeline — from Model Coordinates to Virtual Device Coordinates — to remain device-independent.

The VDC Transform performs this mapping, but it is not directly accessible as an attribute to the application. The VDC Transform maps a rectangular region in VDC known as a *window* to a rectangular region in DC known as a *viewport*. The VDC Transform defaults to the Identity Transform, so VDC is identical to DC. It allows scaling and translation, but not rotation.

XGL provides implicit window resizing policies (although the application must first inform XGL that a window resize event occurred). The window resizing policies automatically recalculate the VDC Transform when a user resizes a Window Raster. The action taken depends on the values of three attributes: XGL_CTX_VDC_MAP, XGL_CTX_VDC_WINDOW, and XGL_CTX_DC_VIEWPORT.

- XGL_CTX_VDC_MAP determines the VDC mapping method (which is also known as the window resizing policy, although it applies to Devices other than Window Rasters). It also specifies whether the application provides the mapping from VDC-to-DC, or if XGL calculates the mapping. It has four values:

  XGL_VDC_MAP_OFF

  This value gives the application complete control over the VDC-to-DC mapping. The application must specify XGL_CTX_VDC_WINDOW and XGL_CTX_DC_VIEWPORT. This means that the application must know the dimensions of the Device (which a user could change if the Device is a Window Raster), and the value of the attribute

XGL_CTX_VDC_ORIENTATION, which provides the correct regions for mapping. (See the next section on VDC Orientation for more information).

XGL_VDC_MAP_ALL
: This value allows the application to provide the value of XGL_CTX_VDC_WINDOW. XGL sets the value of XGL_CTX_DC_VIEWPORT to the Device's bounding box. This method ensures use of the full range of Device Coordinate Space; however, it may result in the image being scaled by different increments along the *x*- and *y*-axes.

XGL_VDC_MAP_ASPECT
: Like the previous value, this value lets the application provide the value XGL_CTX_VDC_WINDOW, but XGL determines the value of XGL_CTX_DC_VIEWPORT. This method maps the VDC window to DC so that the region is as large as possible while still maintaining the aspect ratio of width to height. The top left corner of the XGL_CTX_VDC_WINDOW is mapped to the top left corner (0,0) of the Device.

XGL_VDC_MAP_DEVICE
: For this value, XGL ignores XGL_CTX_VDC_WINDOW and sets the value of XGL_CTX_DC_VIEWPORT to the Raster bounding box. Here XGL maps VDC directly onto DC. The *x*-axis points to the right, and one unit of distance in VDC is one unit in DC. The *y*- and *z*-axes may have opposite directions, as described in the next section, VDC Orientation.

The default value for XGL_CTX_VDC_MAP is XGL_VDC_MAP_DEVICE. If the VDC mapping method is XGL_VDC_MAP_DEVICE, the DC viewport is set to the device maximum coordinates when a Device is attached to a Context.

- XGL_CTX_VDC_WINDOW is the VDC rectangular region to be transformed into DC. The default value for XGL_CTX_VDC_WINDOW is [-1,1] × [-1,1] for 2D Contexts and [-1,1] × [-1,1] × [0,1] for 3D Contexts. This attribute is ignored if the VDC mapping method is XGL_VDC_MAP_DEVICE.
- XGL_CTX_DC_VIEWPORT is the DC rectangular region into which the VDC window maps. If the application attempts to set the value of XGL_CTX_DC_VIEWPORT so that the DC viewport extends partly or entirely outside the Device's boundaries, XGL will not accept this value but will set XGL_CTX_DC_VIEWPORT to the entire extent of the device. This is usually not the desired effect, so applications should detect this condition and take the following actions. If the DC viewport is partially outside the Device's

boundaries, the application should reduce the size of the DC viewport so it is entirely inside the Device, and reduce the VDC window by a proportional amount. If the DC viewport is entirely outside the Device's boundaries, the application can set XGL_CTX_RENDERING to FALSE so that the primitives will not draw into the Device.

The default value for XGL_CTX_DC_VIEWPORT is $[0,M_x] \times [0,M_y] \times [0,M_z]$ where (M_x, M_y, M_z) are the maximum dimensions of the device (see XGL_DEV_MAXIMUM_COORDINATES). However, the value of XGL_CTX_DC_VIEWPORT changes to match a Device when it is associated with the Context.

VDC Orientation

As discussed in the previous section, the orientation of VDC (specified by the attribute XGL_CTX_VDC_ORIENTATION) affects the mapping from VDC to DC. Since the VDC mapping (defined by XGL_CTX_VDC_MAP) only maps a rectangular region of VDC into a rectangular region in DC, the orientation of VDC is the same as DC. However, some applications may prefer to have a VDC space with the y-axis going from bottom to top.

On a workstation screen, DC space is right-handed: the x-axis is positive to the right, the y-axis positive top to bottom, and the z-axis positive away (into the screen). But the interpretation of *up, down, near* and *far* (specified by the orientation of the y- and z-axes) can be altered. This capability is provided by the attribute XGL_CTX_VDC_ORIENTATION. This attribute can take two values:

XGL_Y_DOWN_Z_AWAY

XGL_Y_UP_Z_TOWARD

These values are illustrated in Figure 10-3 on page 302. The default value is XGL_Y_DOWN_Z_AWAY.

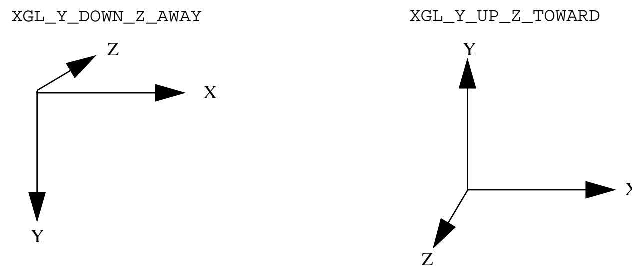


Figure 10-3 VDC Orientation

The Context attribute XGL_CTX_VDC_ORIENTATION controls the interpretation of the coordinates in the VDC system. Application programmers should keep the value of this attribute in mind when they decide how to compute the View Transform. Changing the value of XGL_CTX_VDC_ORIENTATION without changing XGL_CTX_VIEW_TRANS does not make sense because of resulting inconsistency.

MC-to-DC Transform

With the MC-to-DC Transform, an application can find the device coordinate values of a point in model space. The MC-to-DC Transform results from the concatenation of the Model Transform, the View Transform, the VDC Transform, and the Device's DC orientation.

The XGL_CTX_MC_TO_DC_TRANS attribute is read-only. The MC-to-DC Transform can only be changed by modifying the Local Model Transform, the Global Model Transform, the View Transform, or by setting one of the attributes that XGL uses to calculate the VDC-to-DC transformation: XGL_CTX_VDC_MAP, XGL_CTX_VDC_ORIENTATION, XGL_CTX_VDC_WINDOW, and/or XGL_CTX_DC_VIEWPORT. Changing XGL_CTX_MC_TO_DC_TRANS directly with Transform operators is not allowed. The default value of the MC-to-DC Transform is the Identity Transform.

Note – Evaluation of the MC-to-DC Transform is deferred until either XGL needs the Transform internally or the application explicitly gets it. Before attempting to use the MC-to-DC Transform, call `xgl_object_get()` to get `XGL_CTX_MC_TO_DC_TRANS` to ensure that XGL evaluates the MC-to-DC Transform if it requires updating.

Model Clipping in 3D

Model clipping in the 3D pipeline is provided to enable clipping of graphics primitives before viewing. Each model clipping plane is defined by a pair of values given in World Coordinates: a point in the plane and a unit vector normal to the plane. The direction of the vector points in the direction of a half-space whose boundary is the model clipping plane. All the geometry on the side of the normal vector is kept, and the rest is removed by clipping. A clipping volume is defined by intersecting several half-spaces defined by their planes.

When performing model clipping, XGL considers all the enabled model clipping planes in the order provided by the application. By default, XGL does not perform any model clipping.

Model clipping of some primitives, such as annotated primitives, is performed only on the reference point. If the reference point is clipped, the entire primitive is considered clipped.

Model clipping of primitives defined with a point type that lacks an edge flag field will generate a clipped primitive with all edges visible, including those on the model clipping plane boundaries.

The two attributes that define model clipping in the view pipeline are `XGL_3D_CTX_MODEL_CLIP_PLANES`, which specifies the list of model clip planes to use, and `XGL_3D_CTX_MODEL_CLIP_PLANE_NUM`, which specifies the number of model clipping planes in use. When new model clipping planes are set, the application must first specify the number of clip planes with the `XGL_3D_CTX_MODEL_CLIP_PLANE_NUM` attribute and then the list of model clipping planes with `XGL_3D_CTX_MODEL_CLIP_PLANES`. When the attribute `XGL_3D_CTX_MODEL_CLIP_PLANE_NUM` is set to 0, model clipping is disabled.

View Clipping

Applications using a 2D transformation pipeline can perform view clipping in VDC against a rectangular boundary: $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$. The default clip area is $[-1,1] \times [-1,1]$.

An application can disable view clipping against selected edges of the clip boundary.

In a 3D transformation pipeline, primitives can be clipped in homogeneous Virtual Device Coordinates against a view clip volume. The application specifies the volume as a rectangular parallelpiped in ordinary Virtual Device Coordinates: $[x_{min}, x_{max}] \times [y_{min}, y_{max}] \times [z_{min}, z_{max}]$. Thus, view clipping is performed in homogeneous VDC against the view clip volume given by

$$\{ wx_{min} \le x \le wx_{max},\ wy_{min} \le y \le wy_{max},\ wz_{min} \le z \le wz_{max},\ w > 0 \}$$

where *w* is the homogeneous coordinate. The default clip volume in ordinary VDC is $[-1,1] \times [-1,1] \times [0, 1]$. An application can disable view clipping against selected planes of the clip volume.

XGL has three attributes for defining the view clipping (clipping of the image between WC Space and VDC Space):

`XGL_CTX_VIEW_CLIP_BOUNDS`
This attribute defines the clipping *area* (for 2D Contexts) or *volume* (for 3D Contexts) in ordinary VDC. XGL converts the specified clipping region to homogeneous VDC, where clipping is performed. The values given by the application for `XGL_CTX_VIEW_CLIP_BOUNDS` need not be the same as those for the `XGL_CTX_VDC_WINDOW` attribute. The clipping planes defined by this attribute can be individually controlled by the attribute `XGL_CTX_CLIP_PLANES`. The default value is $[-1,1] \times [-1,1]$ for 2D, or $[-1,1] \times [-1,1] \times [0, 1]$ for 3D.

`XGL_CTX_CLIP_PLANES`

This attribute allows the application to specify individually which planes are used in view clipping within the current Context. The application can enable any combination of clipping planes by combining flags that represent each plane with bitwise logical-OR. The flags are:

`XGL_CLIP_XMIN`

`XGL_CLIP_XMAX`

`XGL_CLIP_YMIN`

`XGL_CLIP_YMAX`

`XGL_CLIP_ZMIN`

`XGL_CLIP_ZMAX`

The default value is `NULL`: no view clipping planes enabled.

`XGL_3D_CTX_VIEW_CLIP_PLUS_W_ONLY`

An application can specify a 3D Transform in homogeneous space where the geometry is in front of and behind the viewer (the homogeneous coordinates can have both positive and negative *w* values). This attribute tells XGL how to handle the positive and negative *w* values. If the attribute is `FALSE`, the 3D clipping process checks for both positive and negative *w* values. Primitives are clipped in each region. For example, a line segment that has one endpoint in positive *w* and the other in negative *w* will be split into two line segments. If the attribute is `TRUE` (default value), clipping is restricted to positive *w*.

View Example Program

The following code fragment, `view_perspect.c`, is a section of an example program on perspective viewing. It contains the geometric data and the OpenWindows code for setting up a control panel to generate different views of a wireframe cube. To compile the complete program, type `make view`. The complete program includes `view_perspect.c`, `view_main.c`, and `ex_utils.c`. The latter two programs are listed in Appendix B.

Usage Note

In general, an application should not set the Context attributes XGL_CTX_LOCAL_MODEL_TRANS, XGL_CTX_GLOBAL_MODEL_TRANS, and XGL_CTX_VIEW_TRANS. When an application creates a Context, XGL creates these Transforms to implement the Context's transformation pipeline. An application can get the values of these attributes. Then both the application and the Context, which owns the Transform objects, have a handle to a Transform in the Context's pipeline. If this handle is the destination parameter of a Transform operator, the Context automatically knows about the change to one of its Transforms. For example, the function `view_calc` in the section of code in `view_perspect.c` gets a handle to the View Transform, which it stores in `view_trans`. It then copies the result of concatenations in `trans` to `view_trans`. Figure 10-4 shows the output of `view_perspect.c`.

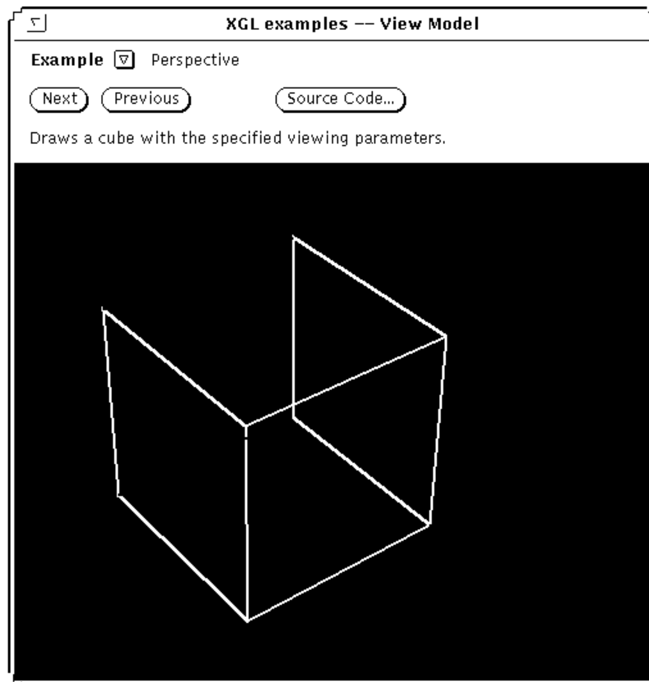


Figure 10-4 Output of `view_perspect.c`

Code Example 10-1 View Example

```
/*
 * view_perspect.c
 */
#include <stdio.h>
#include <math.h>
#include <xview/xview.h>
#include <xview/frame.h>
#include <xview/panel.h>
#include <xgl/xgl.h>

#include "ex.h"

#define PI            3.141592654

extern Xgl_pt_list      pls[];
extern Panel_item       eye_x_item,
                        eye_y_item,
                        eye_z_item,
                        field_of_view_item;

static Xgl_object       ctx = NULL;
static Xgl_bounds_d3d   vdc_window = {-1.0,1.0,-1.0,1.0, -1.0,0.0};
static Xgl_vdc_map      vdc_map = XGL_VDC_MAP_ASPECT;

static void             draw_polylines ();
static void             view_calc (Xgl_pt_f3d*, float);


/****
 *
 * view_perspective
 *
 * This is the entry point from the controls on the example panel.  It
 * initializes some VDC parameters and calculates the next view.
 *
 ***/
void
view_perspective (Xgl_object  ctx3d)
{
    Xgl_pt_f3d           eye_pos;
    float                fov;

    ctx = ctx3d;
    xgl_object_set (ctx,
```

```
                                XGL_CTX_VDC_ORIENTATION, XGL_Y_UP_Z_TOWARD,
                                XGL_CTX_VDC_MAP, vdc_map,
                                XGL_CTX_VDC_WINDOW, &vdc_window,
                                XGL_CTX_VIEW_CLIP_BOUNDS, &vdc_window,
                                XGL_CTX_CLIP_PLANES,
                                XGL_CLIP_XMIN | XGL_CLIP_XMAX | XGL_CLIP_YMIN |
                                XGL_CLIP_YMAX | XGL_CLIP_ZMIN | XGL_CLIP_ZMAX,
                                NULL);

    /* use wider lines to make them more visible */
    xgl_object_set (ctx,
                                XGL_CTX_LINE_WIDTH_SCALE_FACTOR, 2.0,
                                NULL);

    /* get eye position and field of view from panel */
    sscanf((char *)panel_get_value (eye_x_item), "%f", &eye_pos.x);
    sscanf((char *)panel_get_value (eye_y_item), "%f", &eye_pos.y);
    sscanf((char *)panel_get_value (eye_z_item), "%f", &eye_pos.z);
    sscanf((char *)panel_get_value(field_of_view_item),"%f",&fov);

    /* calculate viewing and draw wireframe cube */
    view_calc (&eye_pos, fov);
}


/****
 *
 * view_set
 *
 * This is the entry point from the button "Set view" on the panel
 * for view model parameters. It reads the eye position and field
 * of view, then calculates the next view.
 *
 ***/
void
view_set (
    Panel_item              item,
    int                     value,
    Event                  *event)
{
    Xgl_pt_f3d              eye_pos;
    float                   fov;

    sscanf ((char *)panel_get_value (eye_x_item), "%f", &eye_pos.x);
    sscanf ((char *)panel_get_value (eye_y_item), "%f", &eye_pos.y);
    sscanf ((char *)panel_get_value (eye_z_item), "%f", &eye_pos.z);
```

```
    sscanf((char *)panel_get_value(field_of_view_item),"%f",&fov);

    view_calc (&eye_pos, fov);
}


/****
 *
 * view_calc
 *
 * Calculate the View Transform consisting of the
 * orientation given by the eye position and
 * the perspective given by the field of view.
 * Then redisplay the wireframe object.
 *
 ***/
static
void
view_calc (
    Xgl_pt_f3d          *eye,      /* Eye position in WC */
    float                fov)      /* Field of view (degrees) */
{
    Xgl_pt               pt;
    Xgl_pt_f3d           pt_f3d;
    Xgl_object           trans;
    Xgl_object           view_trans;
    Xgl_object           perspective_trans;
    Xgl_matrix_f3d       matrix;
    double               temp;
    float                distance;
    float                near;
    float                far;
    float                x_view_ratio;

    /* Create a 3D floating-point transform */
    trans = xgl_object_create (sys_st, XGL_TRANS, NULL, NULL);

    /* Set view orientation */

    /* Translate the origin of VDC to the eye position in WC */
    pt.pt_type = XGL_PT_F3D;
    pt.pt.f3d = &pt_f3d;
    pt_f3d.x = -eye->x;
    pt_f3d.y = -eye->y;
    pt_f3d.z = -eye->z;
    xgl_transform_translate (trans, &pt, XGL_TRANS_REPLACE);
```

```
/* Swing the y-axis of VDC so that it is vertical in WC */
xgl_transform_rotate (trans, -PI / 2, XGL_AXIS_X,
                      XGL_TRANS_POSTCONCAT);

/* Swivel the z-axis of VDC away from WC's z-axis */
temp = atan (eye->x / eye->y);
if (eye->y > 0.0)
    temp += PI;
xgl_transform_rotate (trans, temp, XGL_AXIS_Y,
                      XGL_TRANS_POSTCONCAT);

/* Tip VDC so that its z-axis points away from WC's origin */
xgl_transform_rotate (trans, atan (eye->z /
                                   sqrt (eye->x * eye->x +
                                         eye->y * eye->y)),
                      XGL_AXIS_X, XGL_TRANS_POSTCONCAT);

/* Set view perspective and concatenate with view orientation */
distance=sqrt(eye->x *eye->x +eye->y *eye->y +eye->z *eye->z);
near = 2.0 - distance;
far = -2.0 - distance;
x_view_ratio = tan ((fabs (fov) / 2.0) * (PI / 180.0));
matrix[0][0] = 1.0 / x_view_ratio;
matrix[0][1] = matrix[0][2] = matrix[0][3] = 0.0;
matrix[1][1] = 1.0 / x_view_ratio;
matrix[1][0] = matrix[1][2] = matrix[1][3] = 0.0;
matrix[2][2] = 1.0 / (1.0 - near / far);
matrix[2][3] = -1.0;
matrix[2][0] = matrix[2][1] = 0.0;
matrix[3][2] = -near / (1.0 - near / far);
matrix[3][0] = matrix[3][1] = matrix[3][3] = 0.0;
perspective_trans = xgl_object_create (sys_st, XGL_TRANS, NULL,
                                       NULL);
xgl_transform_write (perspective_trans, matrix);
xgl_transform_multiply (trans, trans, perspective_trans);
xgl_object_destroy (perspective_trans);

/* Clear display before changing transformation pipeline */
xgl_context_new_frame (ctx);

/* Set context's view transform */
xgl_object_get (ctx, XGL_CTX_VIEW_TRANS, &view_trans);
xgl_transform_copy (view_trans, trans);

/* Redisplay with new view transform */
```

```
    draw_polylines ();

    /* Clean up */
    xgl_object_destroy (trans);
}


/****
 *
 * vdc_map_set
 *
 * This is the notification procedure for the
 * VDC map choice stack.
 * Set the VDC map and redisplay the polylines.
 *
 ***/
void
vdc_map_set (
    Panel_item           item,
    int                  val,
    Event               *event)
{
    switch (val) {

        /* Map VDC window to entire window raster */
    case 0:
        xgl_context_new_frame (ctx);
        vdc_map = XGL_VDC_MAP_ALL;
        xgl_object_set (ctx, XGL_CTX_VDC_MAP, vdc_map, NULL);
        draw_polylines ();
        break;

       /* Map VDC window to window raster, but retain aspect ratio */
    case 1:
        xgl_context_new_frame (ctx);
        vdc_map = XGL_VDC_MAP_ASPECT;
        xgl_object_set (ctx, XGL_CTX_VDC_MAP, vdc_map, NULL);
        draw_polylines ();
        break;
    default:
        break;
    }
}
```

```
/****
 *
 * draw_polylines
 *
 * Displays the geometric data as polylines.
 *
 ***/
static
void
draw_polylines ()
{
    xgl_multipolyline (ctx, NULL, 4, pls);
}
```

Text

11

This chapter discusses stoke text and raster text in XGL. The chapter includes information on the following topics:

- Stroke Font object operators and attributes
- Context attributes that define the appearance of stroke text
- Rendering text using bitmap fonts

Overview of Text in XGL

The XGL library provides applications with two ways of displaying text. The application can display stroke text using the XGL Stroke Font object. Stroke text draws characters as a series of vectors (strokes) that can be clipped when rendered onto the display device. XGL stroke text can be rendered in any 3D orientation, or it can be rendered as annotations that are guaranteed to be parallel to the display surface. The Stroke Font object includes text primitives and attributes, and when used with related Context text attributes, stroke text provides a full implementation of font support.

The XGL library also provides support for raster text, which enables the application to display text from fonts composed of bitmap characters. XGL's raster text functionality uses the XGL raster operators `xgl_context_copy_buffer()` and `xgl_image()` to render character bitmaps to the device. Raster text can be rendered as stencil text, in which only

pixels in the font character are set. Raster text can also be rendered as block characters, in which the pixels not in the font character are set to the background color.

The following sections present information on how to use XGL stroke text and raster test and provide an example program illustrating each way of displaying text.

Stroke Font Object

A *font* is a particular size and style of type from a font family. Each font represents a specific character set. The XGL Stroke Font object encapsulates a single font, whose characters can then be used by the Context for rendering text. XGL text primitives can render a string of text composed of one font or composed of distinct fonts built from different character sets. Up to four character sets can be active simultaneously. Character sets are made accessible to the Context object by associating Stroke Font objects with the Context.

XGL provides two text encoding schemes. The application programmer can specify ISO encoding or multi-byte encoding (also known as EUC or Extended UNIX Code). Multi-byte encoding enables the application to display characters that require more than one byte per character, such as Kanji. In addition, multiple fonts are allowed per text string. Allowing different encoding schemes and up to four fonts per Context object supports internationalization extensions, enables applications to switch between different languages (character sets) within a text string.

Stroke Fonts Provided by XGL

The XGL library provides a set of stroke fonts that the application can use. At runtime, the XGL library looks for the stroke fonts in the font directory specified by the System State attribute `XGL_SYS_ST_FONT_DIRECTORY`. By default, XGL looks for fonts in `$XGLHOME/lib/xglfonts/stroke` if the `XGLHOME` environment variable is set or in `/opt/SUNWits/Graphics-sw/xgl/lib/xglfonts/stroke` if `XGLHOME` is not set. (For information on the `XGLHOME` environment variable, see page 15.) The application can change the search path for the font directory by setting the attribute `XGL_SYS_ST_FONT_DIRECTORY` to a different path. For more information on the `XGL_SYS_ST_FONT_DIRECTORY` attribute, see Chapter 3,

"System State Information and Generic Operators". XGL fonts are named *fontname.font*. If the application does not plan to use a particular font, the font can be removed from disk to save space.

The fonts that XGL provides are listed Table 11-1. The font names reflect variations on the number and complexity of the strokes used to draw the font characters. Characters in *fontname*_D or *fontname*_T fonts are drawn with more strokes and look a little more substantial than *fontname* fonts. A *fontname*_M font is a monospaced font. Font characters can also be drawn with complex segments, or serifs (*fontname*_C or *fontname*_G).

Table 11-1 Fonts Provided by XGL

	XGL Fonts	
Cartographic	Headline	Roman_C
Cartographic_M	Italic_C	Roman_D
English_G	Italic_T	Roman_M
Greek	Miscellaneous	Roman_T
Greek_C	Miscellaneous_M	Script
Greek_M	Roman	Script_C

Creating a Stroke Font Object

By default, the Context object renders a monospaced Roman font, which is defined at initialization in character set 0. If an application is using ISO encoding, only character set 0 can be used. If an application is using EUC encoding, it can render text using more than one font by creating up to four Stroke Font objects and associating them with the Context object. A Stroke Font object is created with the operator `xgl_object_create()` with a *type* value of `XGL_SFONT`. To create a Stroke Font object, the `xgl_object_create()` operator also needs information on the name of the font. This information is provided via the `desc` parameter, which points to an *Xgl_obj_desc* structure containing the name of the font.

```
obj_desc.sfont_name = "Italic_T.font";
sfont = xgl_object_create(sys_st, XGL_SFONT, &obj_desc, NULL);
```

The font file is retrieved from the stroke font directory, the location of which is specified by the System State attribute XGL_SYS_ST_SFONT_DIRECTORY. To request an XGL font, use the font name with either no file extension or with a file extension of *.font*. The handle for the Stroke Font object (sfont in the example above) can be used simultaneously by multiple Contexts. A Stroke Font object for a particular font name needs to be created only once.

The Context attribute XGL_CTX_SFONT_*x* (where *x* is 0 through 3) associates the name of the Stroke Font object with the Context object. The default value for character set 0 is Roman_M; the other three character sets have no default value. The association of the Stroke Font object with the Context can be set when the Context is created, as in the example below, or set at a later time using the xgl_object_set() operator.

```
ctx = xgl_object_create(sys_st, XGL_2D_CTX, NULL,
                        XGL_CTX_DEVICE, ras,
                        XGL_CTX_SFONT_0, sfont,
                        XGL_CTX_STEXT_CHAR_HEIGHT, 15.0,
                        XGL_CTX_STEXT_TEXT_COLOR, &sf_color,
                        NULL);
```

Rendering Stroke Text

The XGL text primitives render character strings at a specified position on a 2D plane. When stroke text is rendered, the characters are interpreted from the font file in a coordinate system called text local coordinates. This coordinate system is distinct from the application's coordinate system and is defined by the text position and text direction vectors specified in the primitive call. Figure 11-1 illustrates the text local coordinate system. Stroke characters are transformed to the application's model coordinate system before being rendered.

Stroke text is rendered as a set of polylines. The width of the polylines is fixed, and a pattern cannot be applied to the rendered text. Depth cueing is applied to stroke text, but lighting and shading is not applied. Two text primitives are provided for rendering: a primitive to render general stroke text, and a primitive to render annotation text.

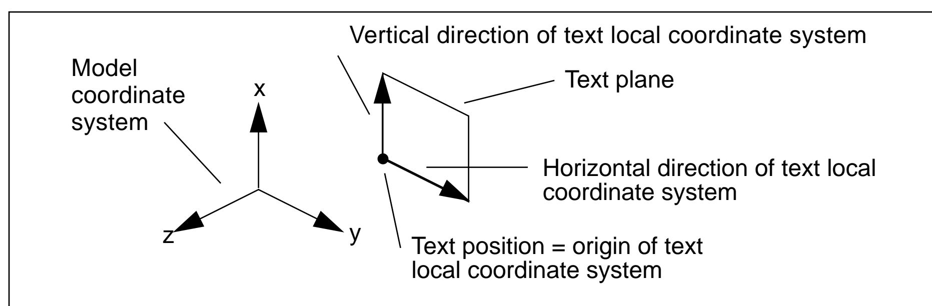


Figure 11-1 Text Local Coordinate System

Rendering Text in 3D Space

The `xgl_stroke_text()` operator renders a string indexed by the characters in the argument `str`. `xgl_stroke_text()` renders text in any orientation in 3D space. The operator is defined as:

```
void xgl_stroke_text (
    Xgl_ctx             ctx,
    void                *str,
    Xgl_pt_f*d          *pos,
    Xgl_pt_f3d          dir[] )
```

`str` is a `NULL`-terminated C-style list of characters if the character encoding selected is `XGL_SINGLE_STR`, or a pointer to a structure of type ***Xgl_mono_text_list***, if the character encoding is `XGL_MULTI_STR`. The parameter `pos` is the 2D or 3D reference point for the position of the rendered string and for the origin of the text plane. `dir` is used for 3D Contexts only; it is an array containing the two direction vectors used for the orientation of the 2D plane on which the text sits. The first vector defines the horizontal direction of the text plane. The second vector specifies the vertical direction and defines an orientation for the text. For 2D Contexts, the text plane is the *x*-*y* plane of the model coordinate system.

Rendering Annotation Text Parallel to the Display Surface

The `xgl_annotation_text()` operator renders a string of text parallel to the display surface.

```
void xgl_annotation_text(
    Xgl_ctx                  ctx,
    void                     *str,
    Xgl_pt_f*d               *ref_pos,
    Xgl_pt_f*d               *anno_pos)
```

`str` is a `NULL`-terminated C-style list of characters if the character encoding selected is `XGL_SINGLE_STR`, or a pointer to a structure of type ***Xgl_mono_text_list***, if the character encoding is `XGL_MULTI_STR`. The argument `ref_pos` specifies the reference point for the text string in Model Coordinates. The text is actually displayed at the annotation point. `anno_pos` defined in VDC is added to the transformed reference position to obtain the annotation point. If the Context attribute `XGL_CTX_ATEXT_STYLE` has been set to `XGL_ATEXT_STYLE_LINE`, a leader line is rendered between the reference point and the annotation point. A leader line connects the reference point to the annotation point and is used to refer to specific parts of an illustration. A leader is shown in Figure 11-2.

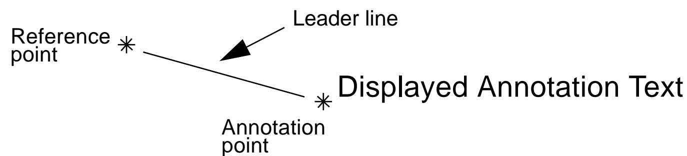


Figure 11-2 Annotation Text and Leader Lines

For annotation text, the text local coordinate system is defined at the annotation point. The *z* value of annotation text is dependent on the *z* value of the annotation point. The text plane is the *x*-*y* plane passing through the *z* coordinate of the annotation point.

Annotation text is clipped if the reference point is outside the clipping region or volume in 3D. If the reference point implies visible text, the text is clipped to the limits of the clipping region or volume.

Single String and Multiple String Rendering

Single string encoding is the simplest way of handing text to XGL. A single C-style (NULL-terminated) text string is passed to the text functions. The attribute XGL_SINGLE_STR uses the font from the associated Context object, and the string of characters that is rendered is a normal C string.

The attribute XGL_MULTI_STR specifies multiple string encoding. Multiple string encoding represents strings as an array of (font, string) elements. The application provides text as an array of *Xgl_mono_text* structures. The *Xgl_mono_text* structure is defined as:

```
typedef struct {
    Xgl_object      font_obj;
    char            *text;
} Xgl_mono_text;
```

and used within a structure of type *Xgl_mono_text_list*:

```
typedef struct {
    Xgl_usgn32      mono_num;
    Xgl_mono_text   *mono_list;
} Xgl_mono_text_list;
```

XGL scans the array for the (font,string) elements, loads the corresponding font and extracts the characters from the string to be rendered. Each sub-string is concatenated to the previous string, forming a single string of text when rendered. Each string in a particular set is rendered using the Stroke Font object associated with it. There is no limit to the number of sets accepted in this coding scheme. The code fragment below shows single string and multiple string rendering.

```
Xgl_sfont             font1;
Xgl_sfont             font2;
Xgl_obj_desc          font_desc;
Xgl_pt_f2d            pos_2d;
Xgl_sgn32             euc = 0;
Xgl_mono_text_list    textlist;
Xgl_mono_text         text[2];

pos_2d.x = 100.0;
```

```
pos_2d.y = 100.0;
font_desc.sfont_name = "Script";
font1 = xgl_object_create(sys_state,XGL_SFONT,&font_desc,NULL);
font_desc.sfont_name = "Roman_C";
font2 = xgl_object_create(sys_state,XGL_SFONT,&font_desc,NULL);

if (!(euc & XGL_MULTI_STR)) {
   xgl_stroke_text (ctx, "Hello World", &pos, NULL);
} else {
   textlist.mono_num = 2;
   textlist_mono_list = text;
   text[0].font_obj = font1;
   text[0].text = "Hello World";
   text[1].font_obj = font2;
   text[1].text = "This is XGL";
   xgl_stoke_text (ctx, textlist, &pos, NULL);
}
```

Stroke Font Object Attributes

To determine the name of the font associated with a Stroke Font object, the application can use the `XGL_SFONT_NAME` attribute with `xgl_object_get()`. The Stroke Font attribute `XGL_SFONT_COMMENT` can also be used to determine the font name. `XGL_SFONT_COMMENT` returns the comment text associated with the font; currently, this information consists of the name of the font only.

The attribute `XGL_SFONT_DEFAULT_CHARACTER` is used to specify the character that is rendered in place of undefined characters in a font. Specifying a default character, such as a blank space, is recommended when first creating the Stroke Font object with `xgl_object_create()`. The default value for this attribute depends on the specified font and may be set in the font file.

The read-only attribute `XGL_SFONT_IS_MONO_SPACED` indicates whether a Stroke Font object contains a monospaced font. In a monospaced font, all the characters of the font have the same width. Computing the area covered by a string of monospaced characters is simpler than computing the area covered by non-monospaced characters. With non-monospaced fonts, the application must call the text extent operator `xgl_stroke_text_extent()` (discussed on page 329) each time it needs to know the width of a particular text string. With monospaced fonts, the application can determine the area covered by the text string from the character width attribute because all characters in the font have the same width.

Context Stroke Text Attributes

The characteristics of stroke text are specified with Context text attributes. Context attributes set character height, character up vector, character width expansion, character spacing, text color, text path, and text alignment. Fonts are associated with the Context object via `XGL_CTX_SFONT_{0,1,2,3}` attributes. The Stroke Font objects must have been created previously with the operator `xgl_object_create()` before being associated with the Context.

Note – If ISO character encoding is used (as defined by the `XGL_CTX_STEXT_CHAR_ENCODING` attribute), only character set 0 can be attached to the Context object. Characters sets 1, 2, and 3 as defined by the attributes `XGL_CTX_SFONT_{1,2,3}`, only apply in the case of EUC multi-byte encoding.

For stroke text, characters are interpreted from the font file in text local coordinates and then transformed to the application Model Coordinate system. For information on annotation text Context attributes, see page 328.

Character Height

The character height attribute, `XGL_CTX_STEXT_CHAR_HEIGHT`, defines the nominal height of text characters. Character height is specified in the application's Local Model Coordinate system. This attribute globally affects character height, character width, and inter-character spacing; thus, if the character height changes, the character width and inter-character spacing will also change. The default value for this attribute is 100.0. Figure 11-3 illustrates the height of characters as defined within a font.

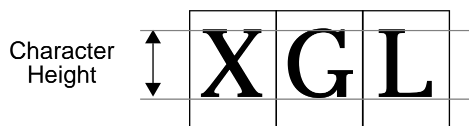


Figure 11-3 Character Height

Character Expansion Factor

The character expansion factor defines the horizontal width of text characters and is specified with the attribute XGL_CTX_STEXT_CHAR_EXPANSION_FACTOR. The character expansion factor modifies the width of characters without changing the character height or the space between characters. The value for this attribute should be positive and is evaluated relative to the character height set by the XGL_CTX_STEXT_CHAR_HEIGHT attribute. The default value of 1.0 renders characters at their normal width.

Character Spacing

The spacing between characters is specified by the XGL_CTX_STEXT_CHAR_SPACING attribute. This attribute allows the application to modify the space between two consecutive stroke text characters without changing the height or width of the characters. Character spacing is expressed as a fraction of the character height specified by the XGL_CTX_STEXT_CHARACTER_HEIGHT attribute. For example, a value of 0.0 adds no additional spacing between characters beyond what is initially defined in the font file; however, a value of 1.0 adds space equal to 1 unit of nominal character height. The default value of this attribute is 0.0. Figure 11-4 shows the default character spacing and a spacing value of 0.5.

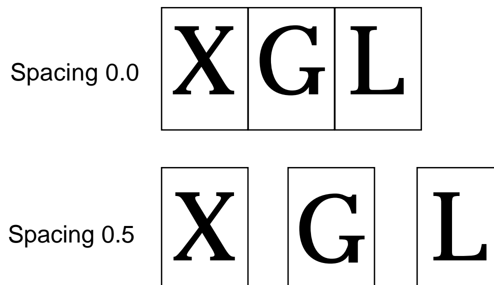


Figure 11-4 Character Spacing

Character Up Vector

The character up vector specifies the *up* direction for text characters and thus controls the orientation of characters. Changing the up vector is equivalent to defining a rotation for the text string. The up vector is defined in text local coordinates and is set with the attribute XGL_CTX_STEXT_CHAR_UP_VECTOR. Figure 11-5 illustrates the default up vector of (0.0, 1.0) and a modified up vector of (0.0,-1.0), which rotates the text string 180 degrees.

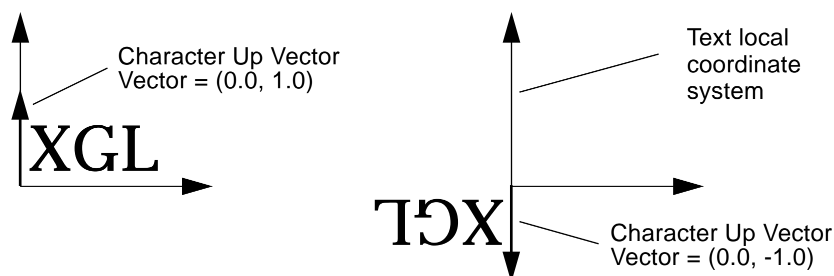


Figure 11-5 Character Up Vector

Note – The character up vector is independent of the VDC orientation defined with the XGL_CTX_VDC_ORIENTATION attribute. XGL automatically sets internal values for the transformation of font characters to the text local coordinate system so that the VDC orientation does not affect rendered text.

Character Slant Angle

Characters can be given an angle with the XGL_CTX_STEXT_CHAR_SLANT_ANGLE attribute. This attribute specifies the angle of inclination of text characters in radians with respect to the upright direction of the text. An angle between - $\pi/2$ and 0 results in an inclination in the backward direction. An angle between 0 and $\pi/2$ results in an inclination in the forward direction. The angle is relative to the text path and the character up vector. The default value for the angle is 0.0. This attribute does not affect text alignment or the height of the characters. Figure 11-6 on page 324 illustrates slant angle.

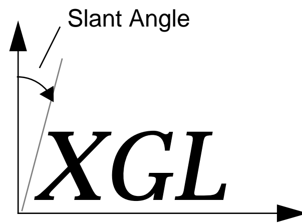


Figure 11-6 Character Slant Angle

Text Path

Text path defines the path or direction that consecutive text characters are rendered relative to the up vector. The text path attribute, `XGL_CTX_STEXT_PATH`, provides four directions to render text from the specified text position. These values are listed below and illustrated in Figure 11-7.

```
XGL_STEXT_PATH_RIGHT
XGL_STEXT_PATH_LEFT
XGL_STEXT_PATH_UP
XGL_STEXT_PATH_DOWN
```

The default value for text path is `XGL_STEXT_PATH_RIGHT`

.

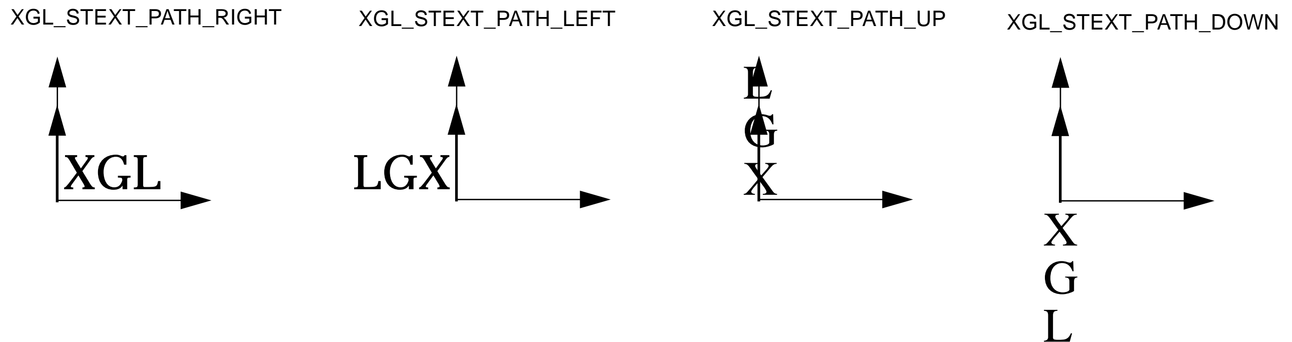


Figure 11-7 Text Path

Text Alignment

Text alignment defines the position of the character string relative to text position. Using a set of horizontal and vertical text alignment attributes, the application can align text in a variety of ways along the horizontal and vertical axes with respect to the text position and the character up vector. The attribute XGL_CTX_STEXT_ALIGN_HORIZ specifies the horizontal alignment.

Along the horizontal axis, the text string can be aligned to the left of the text position, centered at the text position, or aligned to the right of the text position. Values for horizontal alignment are:

```
XGL_STEXT_ALIGN_HORIZ_RIGHT
XGL_STEXT_ALIGN_HORIZ_LEFT
XGL_STEXT_ALIGN_HORIZ_CENTER
XGL_STEXT_ALIGN_HORIZ_NORMAL
```

The attribute XGL_CTX_STEXT_ALIGN_VERT specifies the vertical alignment. The vertical text alignment attributes refer to the internal font lines of a character, which are shown in Figure 11-8.

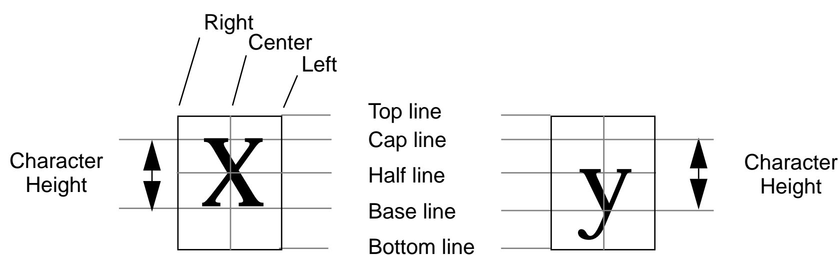


Figure 11-8 Internal Reference Lines of a Character

Along the vertical axis, the text string can be aligned to the top and bottom of the character cell or to the cap, half, and base font lines. Values for vertical alignments are:

```
XGL_STEXT_ALIGN_VERT_BASE
XGL_STEXT_ALIGN_VERT_BOTTOM
XGL_STEXT_ALIGN_VERT_TOP
XGL_STEXT_ALIGN_VERT_HALF
XGL_STEXT_ALIGN_VERT_CAP
XGL_STEXT_ALIGN_VERT_NORMAL
```

Figure 11-9 shows some examples of text alignment for a right horizontal text path. The alignment values are similar for other text paths. The asterisks show the text position for stroke text or the reference position for annotation text.

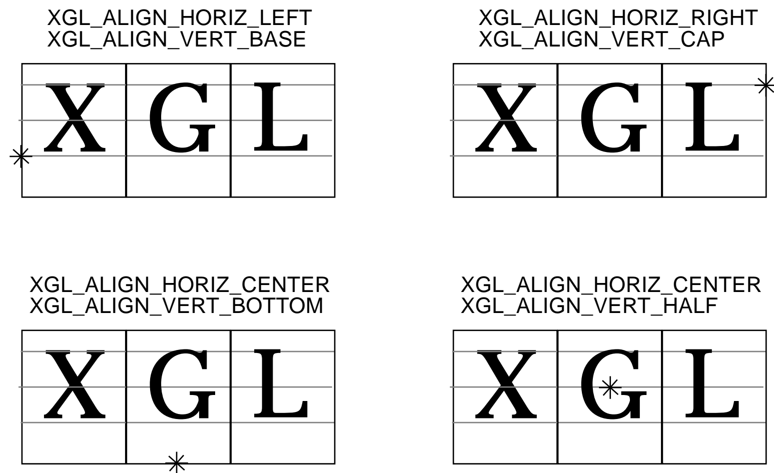


Figure 11-9 Examples of Text Alignment

The default value for `XGL_CTX_STEXT_ALIGN_HORIZ` is `XGL_STEXT_ALIGN_HORIZ_NORMAL`. This value defines the usual horizontal text alignment for each text path. For horizontal text alignment, the normal values are listed in Table 11-2 on page 327.

Table 11-2 Text Horizontal Normal Alignment Values

Text Path	Horizontal Alignment
XGL_STEXT_PATH_RIGHT	XGL_STEXT_ALIGN_HORIZ_LEFT
XGL_STEXT_PATH_LEFT	XGL_STEXT_ALIGN_HORIZ_RIGHT
XGL_STEXT_PATH_UP	XGL_STEXT_ALIGN_HORIZ_CENTER
XGL_STEXT_PATH_DOWN	XGL_STEXT_ALIGN_HORIZ_CENTER

Similarly, the default value for XGL_CTX_STEXT_ALIGN_VERT is XGL_STEXT_ALIGN_VERT_NORMAL. For vertical text alignment, the normal values are listed in Table 11-3.

Table 11-3 Text Vertical Normal Alignment Values

Text Path	Vertical Alignment
XGL_STEXT_PATH_RIGHT	XGL_STEXT_ALIGN_VERT_BASE
XGL_STEXT_PATH_LEFT	XGL_STEXT_ALIGN_VERT_BASE
XGL_STEXT_PATH_UP	XGL_STEXT_ALIGN_VERT_BASE
XGL_STEXT_PATH_DOWN	XGL_STEXT_ALIGN_VERT_TOP

Text Color

The XGL_CTX_STEXT_COLOR attribute defines the color used when rendering stroke text characters. The color must be set to a color index value or an RGB color according to the XGL_RAS_COLOR_TYPE of the Raster associated with the Context. The default value of this attribute is the color of index 1 for indexed devices, or green for RGB devices.

Text Precision

Text precision specifies the minimum precision used when rendering stroke text. It ensures that a precision better than or equal to the precision requested by the application is used when rendering the stroke text. Although his attribute, XGL_CTX_STEXT_PRECISION, has several values, only XGL_STEXT_PRECISION_STROKE is currently supported.

Text precision is related to the way a string of text is clipped when it falls outside the clipping boundaries. XGL_STEXT_PRECISION_STROKE ensures that stroke text is clipped with pixel precision.

Stroke Text Character Encoding

Character encoding can be set with the attribute XGL_CTX_STEXT_CHAR_ENCODING. This attribute specifies which text encoding scheme is used by the application. Two text encoding schemes are currently supported: EUC multi-byte encoding with single or multiple strings or ISO encoding with single or multiple strings. The default value is XGL_SINGLE_STR | XGL_CHAR_MBY. See page 331 for more information on text encoding schemes.

Context Annotation Text Attributes

Annotation text characteristics are specified with Context annotation text attributes. Context attributes set annotation text character height, character up vector, character slant angle, text path, and text horizontal and vertical alignment. A Context annotation text attribute also defines whether annotation text is rendered with a leader line.

Most of the annotation text attributes are identical in function to Context stroke text attributes except that, for annotation text, the characters are transformed to the application Virtual Device Coordinate system rather than the Model Coordinate System. The effect of this is that annotation text is always drawn as if “squared up” to the screen. Table 11-4 on page 329 lists the annotation text attributes and references the page number for the relevant stroke text section for each annotation text attribute. Annotation text style is discussed in the section that follows. Note that for characteristics of annotation text that do not have attributes, the values for the stroke text attributes are automatically used. This is true of character spacing, character encoding, character expansion factor, and text color. Text precision is always stroke precision.

Table 11-4 Annotation Text Attributes

Text characteristic	Annotation text attribute	Page
Character height	XGL_CTX_ATEXT_CHAR_HEIGHT	See page 321.
Character up vector	XGL_CTX_ATEXT_CHAR_UP_VECTOR	See page 323.
Character slant angle	XGL_CTX_ATEXT_CHAR_SLANT_ANGLE	See page 323.
Text path	XGL_CTX_ATEXT_CHAR_PATH	See page 324.
Horizontal text alignment	XGL_CTX_ATEXT_CHAR_ALIGN_HORIZ	See page 325.
Vertical text alignment	XGL_CTX_ATEXT_CHAR_ALIGN_VERT	See page 325.
Annotation text style	XGL_CTX_ATEXT_STYLE	See below.

Annotation Text Style

Annotation text can be rendered in two text styles. The `XGL_CTX_ATEXT_STYLE` attribute displays annotation text at the annotation position with the value `XGL_ATEXT_STYLE_NORMAL`. With the value `XGL_ATEXT_STYLE_LINE`, a leader line links the reference point to the annotation point. The leader line has the current polyline attribute settings. The default value for `XGL_CTX_ATEXT_STYLE` is `XGL_ATEXT_STYLE_NORMAL`.

Determining Text Extent

The text extent is a rectangle that surrounds a text string after the Context stroke text attributes have been applied. Knowing the size of this rectangle can be useful when concatenating text strings. The operator `xgl_stroke_text_extent()` determines the extent of a text string by computing the rectangle in text local coordinates that completely encompasses the text string. The text extent operator is defined as:

```
void xgl_stroke_text_extent (
    Xgl_ctx                  ctx,
    void                     *text,
    Xgl_bounds_f2d           *rect,
    Xgl_pt_f2d               *cat_pt)
```

The argument ctx contains the associated Stroke Font object. str is a pointer to the text string. The operator returns a pointer rect to the rectangular box that surrounds the entire string and returns the concatenation point cat_pt where the string ends in text local coordinates. The concatenation point is the lower right corner of the bounding box. Figure 11-10 shows the text extent rectangle and the concatenation point.

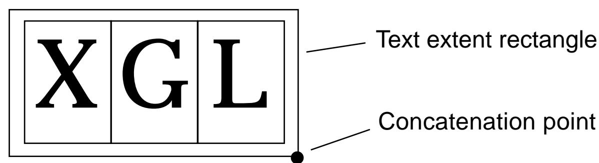


Figure 11-10 Text Extent Rectangle

All Context stroke text attributes except XGL_CTX_STEXT_CHAR_UP_VECTOR and XGL_CTX_STEXT_CHAR_SLANT_ANGLE are used when computing the text extent. This means that when text is drawn rotated at an angle, the concatenation point is calculated from the default up vector.

Text Encoding Schemes

XGL supports two text encoding schemes: ISO encoding and multi-byte encoding. Each of these encoding schemes can be rendered as a string with a single font or as a string with multiple fonts. Thus, an application can render the following combinations of text:

Table 11-5 Text Encoding Schemes

	EUC Encoding	ISO Encoding
Single String Encoding	4 code sets	A single string of an ISO character set, such as English.
Multiple String Encoding	4 code sets, for example Kanji, English, Kanji	A string containing characters from more than one character set, for example English, French, English.

The context attribute `XGL_CTX_STEXT_CHAR_ENCODING` is used to select from the different options by OR-ing together one of each of the following value pairs: `XGL_CHAR_MBY`,`XGL_CHAR_ISO` and `XGL_SINGLE_STR`,`XGL_MULTI_STR`. If an application plans to use only languages with ISO-defined character sets, using EUC encoding is unnecessary.

ISO Encoding

When the application needs to access ISO character sets, ISO encoding must be used. The ISO standard defines 256-entry character sets that contain accentuation and frequently used character symbols. With ISO encoding, the full 8 bits of each character are used to select the character definition. XGL complies with the ISO 8859 normalization. XGL fonts comply with this standard.

ISO strings can be represented as single strings, in which case `XGL_CTX_STEXT_CHAR_ENCODING` is set to `XGL_CHAR_ISO` and `XGL_SINGLE_STR`, and the font used is character set 0 of the Context. ISO can also be represented as multiple strings using the `XGL_CTX_STEXT_CHAR_ENCODING` value `XGL_MULTI_STR`, in which case the font is selected from the array of *Xgl_mono_text* structures passed to the stroke text primitive call.

EUC (Multi-Byte) Encoding:

EUC encoding defines a character-encoding scheme that allows multi-byte characters as well as the mixing of different character sets within a string. EUC encoding uses bit patterns and control characters to specify the different character sets in the string. XGL supports up to four character-encoding sequences, also called *code sets*, which are numbered 0 to 3. EUC encoding uses the most significant bit (MSB) of the character byte and two special characters, SS2 and SS3, to map the characters in the input string into the code sets. The code sets are defined as shown in Table 11-6:

Table 11-6 Code sets in EUC Encoding

Code set	Defined by:
Code set 0	Characters with the MSB set to 0. This is normal 7-bit ASCII character encoding.
Code set 1	Characters with the MSB set to 1.
Code set 2	Characters with the MSB set to 1 and preceded by a single shift byte SS2.
Code set 3	Characters with the MSB set to 1 and preceded by a single shift byte SS3.

In multi-byte encoding, XGL selects the character set used for rendering by decoding the MSB of the byte passed with the character string and determining whether the special shift characters are present. Once the character set is selected, the font associated with the character set can be a single-byte-per-character font or a multiple-byte-per-character font.

As an example of the use of EUC encoding, many Japanese applications use EUC encoding to display English, Kanji, and Katakana (phonetic Japanese) characters within a single string. In this case, the code sets might be associated with character sets as shown in Table 11-7 on page 333:

Table 11-7 Possible use of Code sets in EUC Encoding

Code set	**Used for:**
Code set 0	ASCII
Code set 1	Kanji characters. Each pair of input bytes specifies one Kanji character.
Code set 2	Katakana characters
Code set 3	Implementation-dependent. For example, code set 3 could be used for an additional character set, such as one for special symbols.

For more information on code sets, see the `XGL_CTX_STEXT_CHAR_ENCODING` man page in the *XGL Reference Manual*.

As in the case of ISO encoding, the character string can be coded as a single string, in which case character set 0 of the Context is used, or as multiple strings. For multiple string encoding, the parameters passed to the stroke text call select the font used for the character representations.

Stroke Font Example Program

The example program `stroke_text.c` sets several text attributes and renders a string. Program output is shown in Figure 11-11. To compile this program, type `make stroke_text`. This program includes the main routine and all the window system calls.

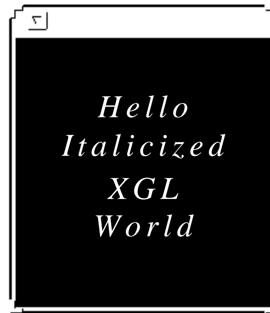


Figure 11-11 Output of `stroke_text.c`

Code Example 11-1 Stroke Text Example

```
/*
 * stroke_text.c.
 */

#include <xview/xview.h>
#include <xview/frame.h>
#include <xview/panel.h>
#include <xview/textsw.h>
#include <xview/canvas.h>
#include <xview/xv_xrect.h>
#include <X11/Xlib.h>
#include <X11/Xatom.h>
#include <X11/Xutil.h>

#include <xgl/xgl.h>

static void             repaint_proc ();
static void             resize_proc ();
static Notify_value     quit_proc (Frame, Destroy_status);
```

```
static Xgl_color_rgb    black_rgb = { 0., 0., 0. };
static Xgl_color_rgb    white_rgb = { 1., 1., 1. };

/* Global variables */
static Xgl_object       ctx = NULL;  /* Context object */
static Xgl_object       ras = NULL;  /* Win ras to write to */
static Xgl_object       sys_st;      /* System state object */
static Xgl_obj_desc     obj_desc;    /* Win ras structure */
static Xgl_pt_f2d      up_vector = {0.0, 1.0}; /* char up vector */

main (
    int                argc,
    char               *argv[])
{
    Frame              frame;   /* XView frame around XGL window */
    Canvas             canvas;  /* XView canvas inside frame */

    Xgl_object         sfont;    /* XGL stroke font object */
    Xgl_color          sf_color; /* stroke font color */
    Xgl_color          bg_color; /* window background color */

    xv_init (XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

/*
 * Create a Frame and a Canvas
 *  Set "repaint_proc" as the procedure to call on repaint events.
 */

    frame = (Frame) xv_create (NULL, FRAME,
                               FRAME_LABEL, "XGL Stroke Text",
                               XV_WIDTH, 220,
                               XV_HEIGHT, 210,
                               NULL);

    canvas = (Canvas) xv_create (frame, CANVAS,
                           XV_X, 0,
                           CANVAS_AUTO_CLEAR,     FALSE,
                           CANVAS_RETAINED,       FALSE,
                           CANVAS_FIXED_IMAGE,    FALSE,
                           CANVAS_REPAINT_PROC,   repaint_proc,
                           CANVAS_RESIZE_PROC,    resize_proc,
                           NULL);

    /* set quit procedure where xgl_close is called */
    (void) notify_interpose_destroy_func(frame, quit_proc);
```

```
{
    Window          frame_window;   /* XID of frame */
    Window          canvas_window;  /* XID of canvas */
    Display         *display;       /* pointer to X display */
    Xv_window       pw;             /* XView paint window */
    Xgl_X_window    xgl_x_win;      /* XGL-X data structure */
    Xgl_inquire     *inq_info;      /* XGL inquiry structure */

    /* get X stuff */
    display = (Display *) xv_get (frame, XV_DISPLAY);

    pw = (Xv_Window) canvas_paint_window (canvas);
    canvas_window = (Window) xv_get (pw, XV_XID);
    frame_window  = (Window) xv_get (frame, XV_XID);

    /* put X stuff into XGL data structure */
    xgl_x_win.X_display = (void *) XV_DISPLAY_FROM_WINDOW (pw);
    xgl_x_win.X_window  = (Xgl_usgn32) canvas_window;
    xgl_x_win.X_screen  = (int) DefaultScreen (display);

/*
 *      Open XGL, create a Raster, a Stroke Font,  and a Context.
 *      Attach the raster as the Context's Device.
 *      Initialize the context text attributes to render the text.
 */
    sys_st = xgl_open (NULL);

    /* use XGL inquiry facility to get hw color type */
    obj_desc.win_ras.type = XGL_WIN_X;
    obj_desc.win_ras.desc = &xgl_x_win;
    if (!(inq_info = xgl_inquire(sys_st, &obj_desc))) {
        printf("error getting inquiry\n");
        exit(1);
    }

    /* set stroke text and background colors appropriately */
    if (inq_info->color_type.index) {
        sf_color.index = 1;
        bg_color.index = 0;
    }
    else if (inq_info->color_type.rgb) {
        sf_color.rgb = white_rgb;
        bg_color.rgb = black_rgb;
    }
    else {  /* get info from visual */
```

```
    XVisualInfo          visual_info;
    int                  visual_class = TrueColor;
    int                  default_screen;
    int                  default_depth;

    /* locate a visual */
    default_screen = DefaultScreen(display);
    default_depth = DefaultDepth(display, default_screen);
    while (!XMatchVisualInfo(display, default_screen,
           default_depth, visual_class--, &visual_info));

    if (visual_info.class == TrueColor) {
        sf_color.rgb = white_rgb;
        bg_color.rgb = black_rgb;
    }
    else {
        sf_color.index = 1;
        bg_color.index = 0;
    }
}

/* user must free memory allocated by inquiry function */
free(inq_info);

/* create raster */
obj_desc.win_ras.type = XGL_WIN_X;
obj_desc.win_ras.desc = &xgl_x_win;
ras = xgl_object_create (sys_st, XGL_WIN_RAS, &obj_desc,
                      NULL);

/* create stroke font */
obj_desc.sfont_name = "Italic_T.font";
sfont = xgl_object_create (sys_st, XGL_SFONT, &obj_desc,
                         NULL);

ctx = xgl_object_create (sys_st, XGL_2D_CTX, NULL,
               XGL_CTX_DEVICE, ras,
               XGL_CTX_DEFERRAL_MODE, XGL_DEFER_ASAP,
               XGL_CTX_SFONT_0, sfont,
               XGL_CTX_STEXT_CHAR_HEIGHT, 15.0,
               XGL_CTX_STEXT_CHAR_UP_VECTOR, up_vector,
               XGL_CTX_STEXT_CHAR_SPACING, 0.5,
               XGL_CTX_STEXT_COLOR, &sf_color,
               XGL_CTX_LINE_COLOR, &sf_color,
               XGL_CTX_BACKGROUND_COLOR, &bg_color,
               NULL);
```

```
    }
/*
 * Let the window system handle the events.
 */
    xv_main_loop (frame);
}

static void
resize_proc ()
{
    if (ras)
        xgl_window_raster_resize(ras);
}

static void
repaint_proc ()
{
    static Xgl_pt_f2d   text_pos;

    if (ctx) {
        xgl_window_raster_resize(ras);
}

static void
repaint_proc ()
{
    static Xgl_pt_f2d   text_pos;

    if (ctx) {

        xgl_window_raster_resize(ras);

        /* clear the display */
        xgl_context_new_frame (ctx);

        /* draw the text string */
        text_pos.x = 60.0, text_pos.y = 50.0;
        xgl_stroke_text (ctx, "Hello", &text_pos, NULL);
        text_pos.x = 20.0, text_pos.y = 80.0;
        xgl_stroke_text (ctx, "Italicized", &text_pos, NULL);
        text_pos.x = 68.0, text_pos.y = 110.0;
        xgl_stroke_text (ctx, "XGL", &text_pos, NULL);
        text_pos.x = 52.0, text_pos.y = 140.0;
        xgl_stroke_text (ctx, "World", &text_pos, NULL);
    }
}
```

```
static Notify_value
quit_proc(
    Frame               frame,
    Destroy_status      status)
{
    if (status == DESTROY_CHECKING)
        xgl_close(sys_st);

    return(notify_next_destroy_func(frame, status));
}
```

Raster Text

XGL raster text functionality enables applications to render text with bitmap fonts. Bitmap fonts are composed of characters specified in bitmaps. Applications can generate bitmap fonts through the X window system, as shown in the example program on page 342, or they can use their own bitmap fonts.

To render raster text, the application must first cache bitmap character information in XGL Memory Raster objects. It can then render the Memory Rasters to the device using the existing XGL raster operators `xgl_context_copy_buffer()` and `xgl_image()`.

Raster text can be rendered as stencils or in block format. Stencil text sets the color of the foreground pixels of the character only; the background pixels are unset and retain their previous color values. Stencil text enables applications to render text on top of an existing image. For raster text in block format, XGL sets the background color of the character bitmap to the current background color. Figure 11-12 shows the difference between stencil text and block text. Raster text is always rendered parallel to the display surface.

Raster text in stencil format

Raster text in block format

Figure 11-12 Stencil Text and Block Text

Caching Font Information

The application is responsible for obtaining and managing bitmap information for font characters. The raster text example program shows how this can be done. Note these guidelines for defining bitmap information:

- Application-defined bitmaps are declared as unsigned short (`Xgl_usgn16`) rather than unsigned char so that the data is properly aligned. In XGL 1-bit Memory Raster format, the pixel row begins at an unsigned short 16-bit boundary with the most significant bit first and padding added to the least significant bits at the end of a pixel row. For information on XGL Memory Raster format, see the `XGL_MEM_RAS_IMAGE_BUFFER_ADDR` man page.
- The application must determine the bitmap size. If Xlib is the source of the font glyphs, X bitmap size parameters can be computed from `XCharStruct` members as follows:

```
width = ((char_info.rgearing - char_info.lbearing) + 15)>>3;
height = char_info.ascent + char_info.descent;
```

  If `XDrawString()` is used to generate the bitmap, the origin of the character should be set to `(-char_info.lbearing, char_info.ascent)`.

When the bitmaps are stored, the application must convert each character bitmap or a string of character bitmaps to an XGL Memory Raster. The Memory Raster must have a depth of 1-bit, and the width and height of the bitmap must be included in the Memory Raster create call, as shown below.

```
font_ras = xgl_object_create (ctx, XGL_MEM_RAS, 0,
                    XGL_RAS_DEPTH, 1,
                    XGL_RAS_WIDTH, width,
                    XGL_RAS_HEIGHT, height,
                    XGL_MEM_RAS_IMAGE_BUFFER_ADDR, &raster,
                    NULL);
```

Note that when creating a Memory Raster, the `XGL_RAS_DEPTH`, `XGL_RAS_WIDTH`, and `XGL_RAS_HEIGHT` attributes must be set before the `XGL_MEM_RAS_IMAGE_BUFFER_ADDR` attribute.

Note – XGL provides only the lowest level of support for stencil functionality. Source rasters for bitmaps can be only 1-bit deep; with rasters of other depths, stencil functionality may be undefined.

Rendering Raster Text

The application can render raster text using XGL raster copy primitives to copy the character or string image into the frame buffer. The text can be rendered as block or stencil. For block text, XGL renders the foreground color of the character and uses the current Context background color as defined in `XGL_CTX_BACKGROUND_COLOR`.

For stencil text, XGL uses the color of `XGL_CTX_SURF_FRONT_COLOR` for the set bits in the character and does not change the color of the unset bits. To enable the stencil feature, set the Context attribute `XGL_CTX_RASTER_FILL_STYLE` to the value `XGL_RAS_FILL_STENCIL`.

The raster primitives render text as follows:

- `xgl_context_copy_buffer()` – Renders Memory Rasters in DC space. This primitive provides better performance than `xgl_image()`.
- `xgl_image()` – Renders Memory Rasters in 2D or 3D space. This operator takes into account the `XGL_3D_CTX_HLHSR_MODE` setting when rendering, enabling text to be rendered with Z-buffer support. This provides for depth-cued text.

Note – Because `xgl_image()` renders data in modeling space, the origin of the bitmaps is provided as 3D floating point values. When XGL transforms the value of this point to 2D DC space, there is a small amount of round-off error introduced. For this reason, strings of characters rendered as individual bitmaps may have one or two pixels between the characters. Therefore, instead of using individual characters as glyphs in 3D, the application should cache the complete sentence as a Memory Raster and render it as a whole using `xgl_image()`.

Raster Text Example Program

The following example program, `raster_text.c`, and its utility file, `raster_text_utils.c`, show how to render X fonts using XGL raster text. The program first sets up the window information and intializes the XGL Context. It then loads five X fonts and converts a range of font characters to XGL Memory Raster objects. Each range of characters becomes an array of XGL Memory Rasters. The program then enters the event loop and draws the text. To compile the program, type `make raster_text` in the example program directory.

Note – You may want to use the `xfd` command to see the font glyphs that you plan to use. The command `xfd -box -fn` *fontname* displays a complete font.

Code Example 11-2 Raster Text Example

```
/*
 * This program shows how to use XGL's XGL_RAS_FILL_STENCIL
 * option for the XGL_CTX_RASTER_FILL_STYLE to render raster text
 * for various X fonts.
 */

#include <stdio.h>
#include <X11/X.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <xgl/xgl.h>

Display       *display;
Window        window;
int           screen;

/* Information needed per glyph (character) */
typedef struct {
    short       lbearing;/* origin to left edge of char */
    short       rbearing;/* origin to right edge of char */
    short       width;/* advance to next char's origin */
    short       ascent;/* baseline to bottom edge of raster */
} charinfo;

/* Information needed by XGL per font */
typedef struct {
    Xgl_object       *ras_obj;
```

```
    charinfo          *cinfo;
    unsigned char     *img;
} rasinfo;

/ * Utility functions defined in raster_text_util.c */
void            Display_string();
void            Load_n_convert();

rasinfo         t36i, h56r, h24i, h24r, f9r;

/*
 * Colors used in this program
 */
Xgl_color       background_color, red_color, green_color, blue_color;
Xgl_color       yellow_color, cyan_color, magenta_color, white_color;
Xgl_color_rgb    background_rgb =        { 0.2, 0.2, 0.2 };
Xgl_color_rgb    red_rgb =               { 1.0, 0.0, 0.0 };
Xgl_color_rgb    green_rgb =             { 0.0, 1.0, 0.0 };
Xgl_color_rgb    blue_rgb =              { 0.0, 0.0, 1.0 };
Xgl_color_rgb    yellow_rgb =            { 1.0, 1.0, 0.0 };
Xgl_color_rgb    cyan_rgb =              { 0.0, 1.0, 1.0 };
Xgl_color_rgb    magenta_rgb =           { 1.0, 0.0, 1.0 };
Xgl_color_rgb    white_rgb =             { 1.0, 1.0, 1.0 };

/ * Fonts used in this program */
#define T36iFont "-adobe-times-bold-i-normal--42-0-0-0-p-0-iso8859-
1"
#define H56rFont "-adobe-helvetica-bold-r-normal--56-0-0-0-p-0-
iso8859-1"
#define H24iFont "-adobe-helvetica-bold-i-normal--24-0-0-0-p-0-
iso8859-1"
#define H24rFont "-adobe-helvetica-bold-r-normal--24-0-0-0-p-0-
iso8859-1"
#define F9rFont "-misc-fixed-medium-r-normal--9-80-100-100-c-60-
iso8859-1"

/* This routine draws the string for the given font */
void
Draw_text (Xgl_object  ctx)
{
    Xgl_pt_i2dras_pos;

    /* clear the display */
    xgl_context_new_frame(ctx);

    /* set the text color */
```

```
 xgl_object_set (ctx,
         XGL_CTX_SURF_FRONT_COLOR, &blue_color, 0);
 /* set the reference(lower right) position for the string */
 ras_pos.x = 20; ras_pos.y = 52;

 /* Rendered the string */
Display_string(ctx, t36i.cinfo, t36i.ras_obj, "== STEAK SALE ==",
                      &ras_pos, " ");
 xgl_object_set (ctx,
         XGL_CTX_SURF_FRONT_COLOR, &cyan_color, 0);
 ras_pos.x = 18; ras_pos.y = 50;
Display_string(ctx, t36i.cinfo, t36i.ras_obj, "== STEAK SALE ==",
                      &ras_pos, " ");

 xgl_object_set (ctx,
         XGL_CTX_SURF_FRONT_COLOR, &yellow_color, 0);
 ras_pos.x = 86; ras_pos.y = 143;
 Display_string(ctx, h56r.cinfo, h56r.ras_obj, "50% OFF",
               &ras_pos, " ");
 ras_pos.x = 327; ras_pos.y = 119;
 Display_string(ctx, h24r.cinfo, h24r.ras_obj, "*",
              &ras_pos, " ");

 ras_pos.x = 83; ras_pos.y = 140;
 xgl_object_set (ctx,
         XGL_CTX_SURF_FRONT_COLOR, &red_color, 0);
 Display_string(ctx, h56r.cinfo, h56r.ras_obj, "50% OFF",
              &ras_pos, " ");

 ras_pos.x = 326; ras_pos.y = 118;
 Display_string(ctx, h24r.cinfo, h24r.ras_obj, "*",
              &ras_pos, " ");

 xgl_object_set (ctx,
         XGL_CTX_SURF_FRONT_COLOR, &green_color, 0);
 ras_pos.x = 23; ras_pos.y = 200;
 Display_string(ctx, h24i.cinfo, h24i.ras_obj, "Choose from:",
              &ras_pos, " ");

 ras_pos.y += 45;
Display_string(ctx, h24r.cinfo, h24r.ras_obj, "~ New York Steak",
                  &ras_pos, " ");
 ras_pos.y += 35;
 Display_string(ctx, h24r.cinfo, h24r.ras_obj, "~ Boneless Filet
                  Mignon Steak", &ras_pos, " ");
 ras_pos.y += 35;
```

```
    Display_string(ctx, h24r.cinfo, h24r.ras_obj, "~ Boneless Top
                      Sirloin Steak", &ras_pos, " ");
    ras_pos.y += 35;
    Display_string(ctx, h24r.cinfo, h24r.ras_obj, "~ Boneless
                      Crossrib Steak", &ras_pos, " ");
    ras_pos.y += 35;
    Display_string(ctx, h24r.cinfo, h24r.ras_obj, "~ T-Bone Steak",
                      &ras_pos, " ");

    xgl_object_set (ctx,
            XGL_CTX_SURF_FRONT_COLOR, &magenta_color, 0);
    ras_pos.y += 45;
    Display_string(ctx, f9r.cinfo, f9r.ras_obj, "* With purchase
                      of a Live Bovine", &ras_pos, " ");
}

main()
{
    XSetWindowAttributes attr;
    int                 depth, i, range;
    Colormap            cmap;
    Visual              *visual;
    XVisualInfo         template;
    Xgl_X_window          x_win;
    Xgl_obj_desc          desc;
    XEvent                event;
    Xgl_sys_state         sys_state;
    Xgl_3d_ctx            ctx;
    Xgl_win_ras           ras;

    /*
     * Base Window setup
     */
    if ((display = XOpenDisplay (NULL)) == NULL) {
        (void) fprintf (stdout, "cannot open display\n");
        exit (1);
    }
    screen = DefaultScreen (display);

    if (XMatchVisualInfo(display, screen, 24, TrueColor,
                          &template)) {
        depth = 24;
        visual = template.visual;
        cmap = XCreateColormap(display, RootWindow(display, screen),
                visual, AllocNone);
    } else {
```

```
    depth = 8;
    visual = DefaultVisual(display, screen);
    cmap = DefaultColormap(display, screen);
}

attr.colormap = cmap;
attr.background_pixel = BlackPixel(display, screen);
attr.border_pixel = BlackPixel(display, screen);
attr.event_mask = ButtonPressMask | ButtonReleaseMask |
        KeyPressMask | ExposureMask;

window = XCreateWindow (display,
    RootWindow(display,screen),
    10, 10,
    400, 450, 0,
    depth,
    InputOutput,
    visual,
    CWEventMask|CWBackPixel|CWBorderPixel|CWColormap,
    &attr);

XMapWindow (display, window);
do {
    XNextEvent (display, &event);
} while (event.type != Expose);

sys_state = xgl_open(NULL);

/* Create XGL raster */
x_win.X_display = display;
x_win.X_window  = window;

desc.win_ras.type = XGL_WIN_X;
desc.win_ras.desc = &x_win;

x_win.X_screen = screen;
ras = (Xgl_object) xgl_object_create (sys_state,
    XGL_WIN_RAS, &desc,
    XGL_DEV_COLOR_TYPE, XGL_COLOR_RGB,
    NULL);

if (!ras)
    fprintf (stderr, "Out of memory, program aborted!\n");

/* initialized color information */
background_color.rgb = background_rgb;
```

```
red_color.rgb = red_rgb;
green_color.rgb = green_rgb;
blue_color.rgb = blue_rgb;
yellow_color.rgb = yellow_rgb;
cyan_color.rgb = cyan_rgb;
magenta_color.rgb = magenta_rgb;
white_color.rgb = white_rgb;

ctx = xgl_object_create (sys_state, XGL_3D_CTX, NULL,
    XGL_CTX_DEVICE, ras,
    XGL_CTX_DEFERRAL_MODE, XGL_DEFER_ASAP,
    XGL_CTX_BACKGROUND_COLOR, &background_color,
    XGL_CTX_NEW_FRAME_ACTION,
      XGL_CTX_NEW_FRAME_CLEAR | XGL_CTX_NEW_FRAME_HLHSR_ACTION,
    XGL_CTX_RASTER_FILL_STYLE, XGL_RAS_FILL_STENCIL,
    NULL);

if (!ctx)
    fprintf (stderr, "Out of memory, program aborted!\n");

/* Converts given X font to XGL raster object */
Load_n_convert (sys_state, display, T36iFont, &t36i, ' ', 'Z');
Load_n_convert (sys_state, display, H56rFont, &h56r, ' ', 'Z');
Load_n_convert (sys_state, display, H24iFont, &h24i, ' ', 'z');
Load_n_convert (sys_state, display, H24rFont, &h24r, ' ', '~');
Load_n_convert (sys_state, display, F9rFont, &f9r, ' ', '~');

Draw_text(ctx);
fprintf(stderr, "\n **** Press any key to quit ****\n");

while (1) {
    XNextEvent (display, &event);

    switch (event.type) {
       case Expose:
       case ButtonPress:
           Draw_text(ctx);
           break;

       case KeyPress:
           /* Clean up allocated memory */
           free (t36i.cinfo);
           free (h56r.cinfo);
           free (h24i.cinfo);
           free (h24r.cinfo);
           free (f9r.cinfo);
```

```
                    free (t36i.img);

                    t36i.img = (unsigned char*)t36i.ras_obj;
                    for (i = 0; i < range; i++, t36i.ras_obj++) {
                        if (*t36i.ras_obj)
                            xgl_object_destroy (*t36i.ras_obj);
                    }
                    free (t36i.img);

                    h56r.img = (unsigned char*)h56r.ras_obj;
                    for (i = 0; i < range; i++, h56r.ras_obj++) {
                        if (*h56r.ras_obj)
                            xgl_object_destroy (*h56r.ras_obj);
                    }
                    free (h56r.img);

                    h24i.img = (unsigned char*)h24i.ras_obj;
                    for (i = 0; i < range; i++, h24i.ras_obj++) {
                        if (*h24i.ras_obj)
                            xgl_object_destroy (*h24i.ras_obj);
                    }
                    free (h24i.img);

                    h24r.img = (unsigned char*)h24r.ras_obj;
                    for (i = 0; i < range; i++, h24r.ras_obj++) {
                        if (*h24r.ras_obj)
                            xgl_object_destroy (*h24r.ras_obj);
                    }
                    free (h24r.img);

                    f9r.img = (unsigned char*)f9r.ras_obj;
                    for (i = 0; i < range; i++, f9r.ras_obj++) {
                        if (*f9r.ras_obj)
                            xgl_object_destroy (*f9r.ras_obj);
                    }
                    free (f9r.img);

                    xgl_object_destroy (ctx);
                    xgl_object_destroy (ras);
                    xgl_close(sys_state);
                    XCloseDisplay (display);
                    return 1;
                }
        }
}
```

The utility program, `raster_text_util.c`, contains routines that convert characters from X fonts into XGL Memory Rasters.

```
/*
 * Raster text utility program: This program shows
 * how to convert and display characters from an X font through XGL.
 *
 * The application needs only two routines, Load_n_convert() and
 * Display_string(), to render raster text under XGL.
 */

#include <X11/Xlib.h>
#include <xgl/xgl.h>

/* Information needed per glyph (character) */
typedef struct {
    short       lbearing;/* origin to left edge of char */
    short       rbearing;/* origin to right edge of char */
    short       width;/* advance to next char's origin */
    short       ascent;/* baseline to bottom edge of raster */
} charinfo;

/* Information needed by XGL per font */
typedef struct {
    Xgl_object          *ras_obj;
    charinfo            *cinfo;
    unsigned char       *img;
} rasinfo;


/* Converts range of characters for the given font to XGL raster
object */
void
font_2_xgl_ras( sys_state, display, font_struct, start, end,
font_ras )
Xgl_sys_state sys_state;
Display *display;
XFontStruct*font_struct;
unsigned int start, end;
rasinfo *font_ras;
{

    int    i,j,k,m;
    GC     graphics_ctxt, gc;
    int    width,height,max_w, max_h;
```

```
int                 fdir, ascent, fdesc;
XCharStruct         char_info;
Pixmap              pixmap;
XGCValues           gcvalues;
XImage              *ximg;
unsigned int        pad;
unsigned char       one_char, *tmp_img;
Xgl_object          *ras_obj = font_ras->ras_obj;
unsigned char       *image = font_ras->img;
charinfo            *cinfo = font_ras->cinfo;

one_char = start;
gcvalues.fill_style=FillSolid;
gcvalues.background=WhitePixel(display,DefaultScreen(display));
gcvalues.foreground=BlackPixel(display,DefaultScreen(display));
gcvalues.font = font_struct->fid;
graphics_ctxt = XCreateGC(display,DefaultRootWindow(display),
          GCForeground | GCBackground |
          GCFont | GCFillStyle, &gcvalues);

gcvalues.foreground=WhitePixel(display,DefaultScreen(display));
gcvalues.fill_style=FillSolid;
gc=XCreateGC(display, DefaultRootWindow(display),
   GCForeground | GCFillStyle,&gcvalues);

max_w = font_struct->max_bounds.rbearing -
                    font_struct->min_bounds.lbearing;
max_h = font_struct->ascent + font_struct->descent;
pixmap = XCreatePixmap(display, DefaultRootWindow(display),
    max_w, max_h,
    DefaultDepth(display, DefaultScreen(display)) );

for (; one_char <= end; one_char++, ras_obj++, cinfo++) {
    /* Clear the pixmap background */
    XFillRectangle(display, pixmap, gc, 0, 0, max_w, max_h);
    /* Get the text extents information */
    XTextExtents(font_struct, &one_char, 1, &fdir, &ascent,
       &fdesc,&char_info);
    width = char_info.rbearing - char_info.lbearing;
    height = char_info.ascent + char_info.descent;
    cinfo->lbearing = char_info.lbearing;
    cinfo->rbearing = char_info.rbearing;
    cinfo->width = char_info.width;
    cinfo->ascent = char_info.ascent;

    /* Skip empty character */
```

```
if ( width == 0 || height == 0 ) {
   *ras_obj = (Xgl_object)NULL;
   continue;
}

tmp_img = image;

/* Render the specified char to pixmap */
XDrawString(display, pixmap, graphics_ctxt,
          -char_info.lbearing,char_info.ascent,
          &one_char, 1 );
/* Now fetch the rendered image from pixmap */
if (!(ximg = XGetImage(display,pixmap,0,0,width,height,1,
                 XYPixmap))) {
   fprintf(stderr,"XGetImage failed! out of memory?\n");
   exit(1);
}

/* Computes how many bytes are needed for each line  */
/* XGL requires the image to be padded to unsigned short */
pad = ((width + 15)>>4) << 1;

/* Just do memcopy() if the X image format
/* confirm with XGL's */
/* Otherwise, get one pixel at a time   */
if (ximg->byte_order == MSBFirst &&
      ximg->bitmap_bit_order == MSBFirst
      && ximg->bitmap_pad >=16 ) {
   for ( i = 0; i < height; i++) {
      memcpy (image, &ximg->data[i * ximg->bytes_per_line],
               pad);
      image += pad;
   }
} else {
   for ( i = 0; i < height; i++) {
      for ( j = 0, m = 0; j < pad; j++, image++) {
          for ( k = 0; k < 7; k++) {
              if ( m < width )
                   *image |= XGetPixel(ximg, m++, i);
              *image <<= 1;
          }
          if ( m < width )
              *image |= XGetPixel(ximg, m++, i);
      }
   }
}
```

```
        /* Create a raster object for each character */
        *ras_obj = xgl_object_create (sys_state, XGL_MEM_RAS, 0,
                XGL_RAS_DEPTH, 1,
                XGL_RAS_WIDTH, width,
                XGL_RAS_HEIGHT, height,
                XGL_MEM_RAS_IMAGE_BUFFER_ADDR, tmp_img,
                NULL);

        if (!*ras_obj)
           fprintf (stderr, "Out of memory, program aborted!\n");

        XDestroyImage (ximg);
    }
    XFreeGC (display, graphics_ctxt);
    XFreeGC (display, gc);
    XFreePixmap (display, pixmap);
}

/* Computes the image size for a range of characters */
unsigned int
compute_mem_size(font_struct, start, end )
XFontStruct     *font_struct;
unsigned int     start, end;
{

    int                 fdir, ascent, fdesc;
    XCharStruct         char_info;
    unsigned int        pad_size, mem_size = 0;
    unsigned char       one_char;

    one_char = start;
    for (; one_char <= end; one_char++) {
        /* Get the text extents information */
        XTextExtents(font_struct, &one_char, 1, &fdir, &ascent,
                      &fdesc,&char_info);

        /* Computes how many shorts are needed for the image     */
        /* XGL requires the image to be padded to unsigned short */
        pad_size = ((char_info.rbearing -
                     char_info.lbearing + 15) >> 4) *
                     (char_info.ascent + char_info.descent);
        mem_size += pad_size;
    }
    return (mem_size << 1);
}
```

```
/* Renders a string for the given XGL font(raster) object */
void
Display_string( ctx, cinfo, font_ras, str, pos, offset)
Xgl_objectctx;
charinfo*cinfo;
Xgl_object*font_ras;
unsigned char*str;
Xgl_pt_i2d*pos;
unsigned char*offset;
{
    int         i,j,x;
    Xgl_pt_i2d  cp_pos;
    int len = strlen(str);

    x = pos->x;
    for ( i = 0; i < len; i++, str++) {
        j = *str - *offset;
        if (font_ras[j]) {
           cp_pos.x = x + cinfo[j].lbearing;
           cp_pos.y = pos->y - cinfo[j].ascent;
           xgl_context_copy_buffer (ctx,NULL,&cp_pos,font_ras[j]);
        }
        x += cinfo[j].width;
    }
}

/*
 * Loads a given X font and converts a range of characters to XGL
 * raster objects
 */
void
Load_n_convert (sys_st, display, font_name, font_ras, start, end)
Xgl_sys_state sys_st;
Display *display;
char    *font_name;
rasinfo *font_ras;
unsigned charstart, end;
{
    unsigned intrange;
    XFontStruct*font_struct;

    font_struct = XLoadQueryFont(display, font_name);
    if (!font_struct) {
        fprintf (stderr, "Cannot load assigned font,
                    progam aborted!\n");
```

```
        exit(1);
    }

    range = end - start + 1;

    /* Allocate memory for the array of character object */
    font_ras->ras_obj = (Xgl_object*)calloc(range,
                                   sizeof(Xgl_object));
    if (!font_ras->ras_obj) {
        fprintf (stderr, "Out of memory, program aborted!\n");
        exit(1);
    }

    /* Allocate memory for the array of character information */
    font_ras->cinfo = (charinfo*)calloc(range, sizeof(charinfo));
    if (!font_ras->cinfo) {
        fprintf (stderr, "Out of memory, program aborted!\n");
        exit(1);
    }

    /*
     * Allocate memory for the bitmap image for the font.
     * Memory must be at least aligned with 16 bit boundary.
     */
    font_ras->img = (unsigned char*)memalign
                     (4, compute_mem_size(font_struct,
                     start, end));
    if (!font_ras->img) {
        fprintf (stderr, "Out of memory, program aborted!\n");
        exit(1);
    }

    /*
     * Create memory raster for the given font and
     * starting and ending chars
     */
    font_2_xgl_ras (sys_st, display, font_struct, start,
                  end, font_ras);

    /* We can free the XFont now */
    XFreeFont (display, font_struct);
}
```

Line Patterns 12

This chapter discusses the Line Pattern object and its associated attributes. It includes information on the following topics:

- Using the line patterns supplied with the XGL library
- Creating new line patterns
- Setting the pattern for surface edges

Introduction to the Line Pattern Object

Line primitives (polylines and B-spline curves) are characterized by their line style and color. Line primitives can be rendered as solid lines, the default line style, or as patterns of on-off segments known as *line patterns*. To define the line pattern of a line or curve, the application can use line patterns supplied by XGL, or it can create its own line patterns using the Line Pattern object. XGL-supplied line patterns are predefined and are available for application use at any time. Line patterns defined by the Line Pattern object are used for rendering when the Line Pattern object is attached to a Context object.

Line patterns are also used to define the appearance of the edges of surface primitives. The default style for a surface edge is solid. The edge pattern can be changed by setting it to one of the XGL-supplied line patterns or to the pattern defined in a Line Pattern object.

Patterns for lines and surface edges can be rendered in a single color or in two colors that alternate by line segments.

Using the Predefined Line Patterns

XGL supplies ten line patterns for applications to use. These predefined line patterns are Line Pattern objects that the System State object creates when XGL is initialized. Predefined Line Pattern objects are read-only objects that cannot be modified or destroyed. The predefined line patterns are shown in Figure 12-1.

Line Style Name	Looks Like:
xgl_lpat_dotted	
xgl_lpat_dashed	
xgl_lpat_dash_dot	
xgl_lpat_dashed_dotted	
xgl_lpat_dash_dot_dotted	
xgl_lpat_long_dashed	
xgl_lpat_cgm_dotted	
xgl_lpat_cgm_dashed	
xgl_lpat_center	
xgl_lpat_phantom	

Figure 12-1 Predefined Line Patterns

To use a predefined line pattern, the application sets the Context attribute `XGL_CTX_LINE_STYLE` to `XGL_LINE_PATTERNED` and sets `XGL_CTX_LINE_PATTERN` to the pattern name, as in the following example:

```
xgl_object_set(ctx, XGL_CTX_LINE_STYLE, XGL_LINE_PATTERNED,
                    XGL_CTX_LINE_PATTERN, xgl_lpat_dashed, NULL);
```

Similarly, to specify patterned surface edges, the application must set the Context attribute XGL_CTX_EDGE_STYLE to XGL_LINE_PATTERNED and set XGL_CTX_EDGE_PATTERN to the line pattern name, as in the following example:

```
xgl_object_set(ctx, XGL_CTX_EDGE_STYLE, XGL_LINE_PATTERNED,
                     XGL_CTX_EDGE_PATTERN, xgl_lpat_dashed, NULL);
```

The default value for the attributes XGL_CTX_LINE_PATTERN or XGL_CTX_EDGE_PATTERN is NULL.

Creating a Line Pattern Object

A Line Pattern object is created with the xgl_object_create() operator, using XGL_LPAT as the *type* parameter value. The data for the line pattern is attached to the Line Pattern object with the XGL_LPAT_DATA attribute, as in the following example:

```
lpat = xgl_object_create(sys_st, XGL_LPAT, NULL,
                              XGL_LPAT_DATA_SIZE, 4,
                              XGL_LPAT_DATA, lpatdata,
                              NULL);
```

Defining the pattern lpatdata for the new Line Pattern object is discussed in the section below.

To render lines, curves, or surface edges in the new pattern, the Line Pattern object must be attached to a Context object using the XGL_CTX_LINE_PATTERN attribute for lines and curves or the XGL_CTX_EDGE_PATTERN attribute for surface edges.

```
xgl_object_set(ctx, XGL_CTX_LINE_PATTERN, lpat, NULL);
```

Defining a New Line Pattern

A line pattern is defined in a line segment data array that is associated with the Line Pattern object. The line segment data array stores line segment lengths of alternating *on* and *off* pattern segments. If the pattern has an even number of line segments, the segments are used cyclically; when the end of a pattern is reached (that is, when the end of the array is reached), the pattern starts over again from the beginning. If the pattern has an odd number of line segments, XGL concatenates two repetitions of the pattern so that an even-length line pattern is created, inverts the *on* and *off* segments for the second half of the concatenated pattern, and then uses this pattern as the line pattern.

The first segment of a line pattern is assumed to be an *on* segment. The pattern wraps around along lines, curves, and corners until the end of the line is reached. The code fragment below illustrates a line pattern data array with four segments.

```
float           lpatdata[4];

lpatdata[0] = 1.0; lpatdata[1] = 3.0;
lpatdata[2] = 10.0; lpatdata[3] = 5.0;
```

Figure 12-2 illustrates line patterns with even and odd numbers of line segments.

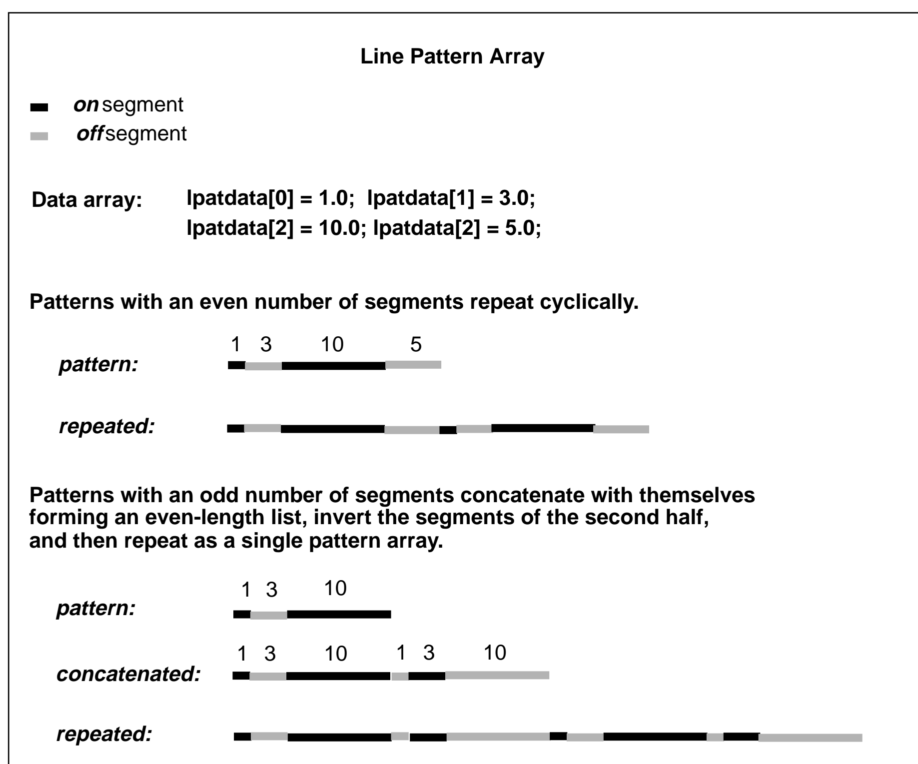


Figure 12-2 Line Pattern Array Formation

Line Pattern Attributes

The pattern of a line or edge pattern is defined with Line Pattern attributes. The Line Pattern attribute XGL_LPAT_DATA defines the on and off segments that comprise a line pattern. The on and off segments are stored as an array of either 32-bit integers or floating-point numbers. The data type used to specify the segments is set with XGL_LPAT_DATA_TYPE; the values for XGL_LPAT_DATA_TYPE are XGL_DATA_INT or XGL_DATA_FLT. The default data type is float.

The length of the line segment data array is specified by the XGL_LPAT_DATA_SIZE attribute, which must be set before passing the data. The default value for XGL_LPAT_DATA_SIZE is 0.

The offset into the line segment data array where the line pattern will begin is specified with the XGL_LPAT_OFFSET attribute. The offset into the line pattern is used only at the beginning of the patterned line being rendered. It is interpreted as the distance into the line pattern at which the pattern will be started. The application should ensure that this attribute is less than the size of the data array specified by XGL_LPAT_DATA_SIZE. The default value for XGL_LPAT_OFFSET is 0.

The way in which a line pattern is applied to a polyline can be changed with the attribute XGL_LPAT_STYLE. This attribute has the following values:

XGL_LPAT_FIXED_OFFSET
: The line pattern is applied with the offset defined by XGL_LPAT_OFFSET. The pattern carries over from one line segment to the next. This is the default behavior.

XGL_LPAT_BALANCED_SEGMENT
: For each segment of a polyline, the line pattern is balanced around the midpoint of the segment before clipping. In this case, the line pattern does not carry over to the adjacent segment; the balancing is independent of the adjacent segments. This attribute is only valid with polylines and does not apply to surface edges.

If XGL_LPAT_STYLE is set to XGL_LPAT_BALANCED_SEGMENT, the application can specify whether to balance the line pattern around the first or second dash using the attribute XGL_LPAT_BALANCED_DASH. This attribute enables the application to center the first dash of the pattern at the midpoint of the segment (XGL_LPAT_BAL_DASH_0) or center the second dash of the pattern at the midpoint of the segment (XGL_LPAT_BAL_DASH_1), as shown in the example in Figure 12-3. XGL_LPAT_BAL_DASH_0 is the default value. The midpoint of a line segment is calculated in device coordinates.

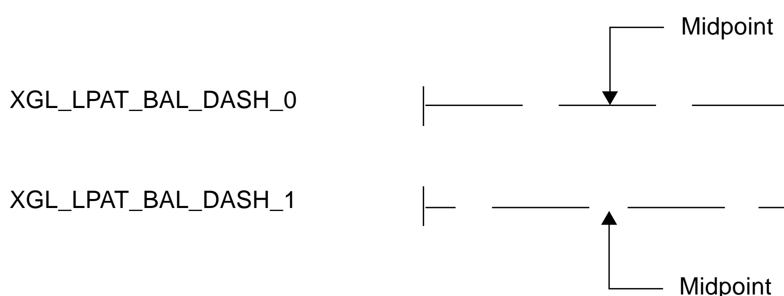


Figure 12-3 Balancing Line Patterns around a Line Segment Midpoint

Context Line Pattern Attributes

The appearance of line and edge patterns when rendered is controlled by Context attributes. The line and edge Context attributes related to the line pattern rendering are discussed below. For more information on the Context line pattern rendering attributes, see the appropriate man pages.

Line Rendering with a Line Pattern

The attribute `XGL_CTX_LINE_STYLE` defines the way in which lines or curves are drawn in a given Context. To render a patterned line, the application must set this attribute to one of the following values:

`XGL_LINE_PATTERNED`
Lines are patterned using the Line Pattern object given by `XGL_CTX_LINE_PATTERN`, and color is specified by the `XGL_CTX_LINE_COLOR` attribute.

`XGL_LINE_ALT_PATTERNED`
Lines are patterned using the Line Pattern object given by attribute `XGL_CTX_LINE_PATTERN`. The color is specified by the attributes `XGL_CTX_LINE_COLOR` for odd line segments and `XGL_CTX_LINE_ALT_COLOR` for even line segments.

`XGL_LINE_SOLID`
Lines are not patterned. A solid line is drawn in the color specified by the `XGL_CTX_LINE_COLOR` attribute.

The default value for the attribute XGL_CTX_LINE_STYLE is XGL_LINE_SOLID.

When the attribute XGL_CTX_LINE_STYLE is set to XGL_LINE_PATTERNED or XGL_LINE_ALT_PATTERNED, the Context attribute XGL_CTX_LINE_PATTERN specifies the line pattern. The attribute XGL_CTX_LINE_PATTERN attaches a Line Pattern object to a Context.

Setting Line Color

The attribute XGL_CTX_LINE_COLOR sets the color of lines and curves when rendering line segments. If the data supplied with polylines has color for the individual vertices, XGL applies these colors to the vertices. If the data does not have color for the vertices, XGL uses the Context line color. The application can use the XGL_CTX_LINE_COLOR_SELECTOR attribute to control whether Context color or vertex color is used for lines with color data at the vertices. XGL_CTX_LINE_COLOR_SELECTOR has the following values:

XGL_LINE_COLOR_CONTEXT
: Line color is rendered as defined by XGL_CTX_LINE_COLOR. Using the Context color to display lines with vertex color may be useful for echoing a picking response. Depth cueing affects the final appearance of the line in 3D.

XGL_LINE_COLOR_VERTEX
: Line color is taken from the vertex data if the vertex data includes color data for individual vertices. If the vertex data does not specify colors for individual vertices, the color defined by XGL_CTX_LINE_COLOR.

The color type (RGB or indexed) must match the color type set by the XGL_RAS_COLOR_TYPE attribute of the Raster associated with the Context. The default value is a color of index 1 for indexed devices, or green for RGB devices. When specifying a color of type XGL_COLOR_INDEX in a 3D Context with depth cueing computations enabled, the color table of the appropriate Color Map object must have consistent color ramps to ensure a correct rendering.

The attribute XGL_CTX_LINE_ALT_COLOR sets the color to fill even (or alternate) line segments when the line style (XGL_CTX_LINE_STYLE) is set to XGL_CTX_LINE_ALT_PATTERNED. The odd line segments are filled using the color specified by XGL_CTX_LINE_COLOR. The color type (RGB or indexed) must match the color type set by the XGL_RAS_COLOR_TYPE attribute of the

Raster associated with the Context. The default value for the attribute XGL_CTX_LINE_ALT_COLOR is a color of index 0 for indexed devices, or black for RGB devices.

Edge Rendering with a Line Pattern

Edges are drawn around the perimeter of surfaces created by primitives capable of edge rendering (such as a polygon). Edges are drawn if the surface edge attribute XGL_CTX_SURF_EDGE_FLAG is set to TRUE. Primitives that accept point data that includes edge flag information can use the edge flag information to turn on or off the rendering of edge boundaries, but this is only applicable if the global edge flag attribute XGL_CTX_SURF_EDGE_FLAG is set to TRUE. The boundaries produced by clipping are not drawn.

The attributes specifying patterned edges are similar to those specifying patterned lines. The attribute XGL_CTX_EDGE_STYLE defines the edge style used for edges of surfaces. The values for this attribute are the same as the values for XGL_CTX_LINE_STYLE:

XGL_LINE_PATTERNED
: Edges are patterned using the Line Pattern object given by XGL_CTX_EDGE_PATTERN. Color is specified by the XGL_CTX_EDGE_COLOR attribute.

XGL_LINE_ALT_PATTERNED
: Edges are patterned using the Line Pattern object given by XGL_CTX_EDGE_PATTERN. The color is specified by the attributes XGL_CTX_EDGE_COLOR for odd line segments and XGL_CTX_EDGE_ALT_COLOR for even line segments.

XGL_LINE_SOLID
: Edges are not patterned. A solid edge is drawn in the color specified by the XGL_CTX_EDGE_COLOR attribute.

The default value for the attribute XGL_CTX_EDGE_STYLE is XGL_LINE_SOLID.

When XGL_CTX_EDGE_STYLE is set to XGL_LINE_PATTERNED or XGL_LINE_ALT_PATTERNED, the attribute XGL_CTX_EDGE_PATTERN attaches a Line Pattern object that specifies the pattern used for edge drawing to a Context. The default value for the attribute XGL_CTX_EDGE_PATTERN is NULL.

Setting Edge Color

The attribute XGL_CTX_EDGE_COLOR sets the color of the surface edges. The color type (RGB or indexed) must match the color type set by the XGL_RAS_COLOR_TYPE attribute of the Raster associated with the Context.

When specifying a color of type XGL_COLOR_INDEX in a 3D Context with depth cueing computations enabled, the color table of the appropriate Color Map object must have consistent color ramps to ensure a correct rendering. The default value is a color of type XGL_COLOR_INDEX with a value of 1.

The attribute XGL_CTX_EDGE_ALT_COLOR sets the color to fill even (or alternate) line segments when the edge style (XGL_CTX_EDGE_STYLE) is set to XGL_CTX_LINE_ALT_PATTERNED. The odd edge segments are filled using the color specified by XGL_CTX_EDGE_COLOR. The color type (RGB or indexed) must match the color type set by the XGL_RAS_COLOR_TYPE attribute of the Raster associated with the Context. The default value is a color of type XGL_COLOR_INDEX with a value of 0.

Line Pattern Examples

The following example programs use the Line Pattern object to create patterned lines and patterned edges. Each example is a fragment of a larger program. The complete program includes ex_utils.c and lpat_main.c, both of which are listed in Appendix B, as well as the two examples listed in this chapter. To compile the complete program, type make lpat in the example program directory. The compiled program allows you to cycle through both line pattern example programs.

Patterned Line Example

Example lpat_lines.c creates three patterns and draws 10 lines for each pattern. The output is shown on Plate 6.

Code Example 12-1 Line Pattern Example

```
/*
 * lpat_lines.c
 */
#include <xview/xview.h>
#include <xgl/xgl.h>
```

```
#include "ex.h"

/*
 * an easy to use macro for setting up an Xgl point list data
 * structure
 */

#define  XGL_SET_PT_LIST_I2D(PL, PT_TYPE, BBOX, NUM_PTS, PTS_I2D)\
           PL.pt_type = PT_TYPE; \
           PL.bbox = BBOX;       \
           PL.num_pts = NUM_PTS; \
           PL.pts.i2d = PTS_I2D;

/*
 * this routine draws 10 lines for each pattern in order to
 * emphasize what the pattern looks like...
 */
void
lpat_lines (Xgl_object     ctx)
{
    Xgl_sgn32            i;
    Xgl_color            ln_fg_color,
                         ln_bg_color;
    Xgl_pt_i2d           pts_i2d[2];
    Xgl_pt_list          pl_2d;

    /*
     * set to dashed line pattern object, set the line style to
     * patterned, set the line foreground pixel color (bits
     * in the pattern that have the value of 1 will be MAGENTA),
     * copy the point data into the Xgl data structure,
     * and then draw the lines
     */
    ln_fg_color = magenta_color;
    xgl_object_set (ctx,
                  XGL_CTX_LINE_PATTERN, xgl_lpat_dashed,
                  XGL_CTX_LINE_STYLE, XGL_LINE_PATTERNED,
                  XGL_CTX_LINE_COLOR, &ln_fg_color,
                  NULL);
    XGL_SET_PT_LIST_I2D (pl_2d, XGL_PT_I2D, NULL, 2, pts_i2d);

    pts_i2d[0].x = 10;
    pts_i2d[0].y = 10;
    pts_i2d[1].x = 410;
    pts_i2d[1].y = 10;
```

```
for (i = 0; i < 10; i++) {
       xgl_multipolyline (ctx, NULL, 1, &pl_2d);
       pts_i2d[0].y++;
       pts_i2d[1].y++;
}

/*
 * set to dotted line pattern object, set the line color
 * to cyan (bits in the pattern that have the value of 1
 * will be CYAN), set line style to alternate line pattern,
 * set the line alternate color to green (bits in the
 * pattern that have a value of 0 will be BLUE),
 * and then draw the lines
 */
ln_fg_color = cyan_color;
ln_bg_color = blue_color;
xgl_object_set (ctx,
               XGL_CTX_LINE_PATTERN, xgl_lpat_dotted,
               XGL_CTX_LINE_STYLE, XGL_LINE_ALT_PATTERNED,
               XGL_CTX_LINE_COLOR, &ln_fg_color,
               XGL_CTX_LINE_ALT_COLOR, &ln_bg_color,
               NULL);

pts_i2d[0].x = 10;
pts_i2d[0].y = 25;
pts_i2d[1].x = 410;
pts_i2d[1].y = 25;
for (i = 0; i < 10; i++) {
       xgl_multipolyline (ctx, NULL, 1, &pl_2d);
       pts_i2d[0].y++;
       pts_i2d[1].y++;
}

/*
 * set to Xgl dashed-dotted line pattern object, set
 * the line color to white, the alternate line color to
 * red and then draw the lines
 * NOTE: still doing alternate line colors
 */
ln_fg_color = white_color;
ln_bg_color = red_color;
xgl_object_set (ctx,
               XGL_CTX_LINE_PATTERN, xgl_lpat_dashed_dotted,
               XGL_CTX_LINE_COLOR, &ln_fg_color,
               XGL_CTX_LINE_ALT_COLOR, &ln_bg_color,
               NULL);
```

```
pts_i2d[0].x = 10;
pts_i2d[0].y = 40;
pts_i2d[1].x = 410;
pts_i2d[1].y = 40;
for (i = 0; i < 10; i++) {
        xgl_multipolyline (ctx, NULL, 1, &pl_2d);
        pts_i2d[0].y++;
        pts_i2d[1].y++;
}

/*
 * set our own dashed line pattern object, set the line
 * style to patterned, set the line foreground pixel color
 * (bits in the pattern with the value of 1 will be WHITE),
 * copy the point data into the Xgl data structure,
 * and then draw the lines
 */
ln_fg_color = white_color;
xgl_object_set (ctx,
              XGL_CTX_LINE_STYLE, XGL_LINE_PATTERNED,
              XGL_CTX_LINE_COLOR, &ln_fg_color,
              NULL);

/*
 * create our own line pattern and draw a line with it
 */
{
        Xgl_sgn32           i;
        Xgl_object          lpat;
        float               lpatdata[6];

        lpatdata[0] = lpatdata[1] = 2.;
        lpatdata[2] = lpatdata[3] = 4.;
        lpatdata[4] = lpatdata[5] = 1.;

        lpat = xgl_object_create (sys_st, XGL_LPAT, NULL,
                    XGL_LPAT_DATA_SIZE, 6,
                    XGL_LPAT_DATA, lpatdata,
                    NULL);

        xgl_object_set (ctx, XGL_CTX_LINE_PATTERN, lpat, NULL);

        pts_i2d[0].x = 10;
        pts_i2d[0].y = 55;
        pts_i2d[1].x = 410;
```

```
                pts_i2d[1].y = 55;
                for (i = 0; i < 10; i++) {
                    xgl_multipolyline (ctx, NULL, 1, &pl_2d);
                    pts_i2d[0].y++;
                    pts_i2d[1].y++;
                }

                xgl_object_set (ctx,
                       XGL_CTX_LINE_STYLE, XGL_LINE_SOLID,
                       NULL);
                xgl_object_destroy (lpat);
          }
      }
```

Line Pattern Polygon Edge Example

Example `lpat_pgon_edges.c` uses the predefined and custom-patterned lines shown in the previous example as polygon edges. The output is shown on Plate 6.

Code Example 12-2 Edge Pattern Example

```
/*
 * lpat_pgon_edges.c
 */

#include <xview/xview.h>
#include <xgl/xgl.h>

#include "ex.h"

extern Xgl_pt_i2d           pts_i2d_pgon1[];
extern Xgl_pt_i2d           pts_i2d_pgon2[];
extern Xgl_pt_i2d           pts_i2d_pgon3[];
extern Xgl_pt_i2d           pts_i2d_pgon4[];

/*
 * an easy to use macro for setting up an Xgl point list data
structure
 */
#define XGL_SET_PT_LIST_I2D(PL, PT_TYPE, BBOX, NUM_PTS, PTS_I2D)\
          PL.pt_type = PT_TYPE;  \
          PL.bbox = BBOX;        \
          PL.num_pts = NUM_PTS;  \
```

```
            PL.pts.i2d = PTS_I2D;

void
lpat_pgon_edges (Xgl_object   ctx)
{
    Xgl_color             ln_fg_color,
                          ln_bg_color,
                          pgon_front_color;
    Xgl_pt_list           pl_2d;

    /*
     * turn those edges on so we can see our patterned edges
     */
    xgl_object_set (ctx, XGL_CTX_SURF_EDGE_FLAG, TRUE, NULL);

    /*
     * set to dashed line pattern object, set the line style to
     * patterned, set the line foreground pixel color (bits
     * in the pattern with the value of 1 will be MAGENTA),
     * copy the point data into the Xgl data structure,
     * and then draw the line
     */
    ln_fg_color = magenta_color;
    pgon_front_color = green_color;
    xgl_object_set (ctx,
                  XGL_CTX_EDGE_PATTERN, xgl_lpat_dashed,
                  XGL_CTX_EDGE_STYLE, XGL_LINE_PATTERNED,
                  XGL_CTX_EDGE_COLOR, &ln_fg_color,
                  XGL_CTX_SURF_FRONT_COLOR, &pgon_front_color,
                  NULL);

    XGL_SET_PT_LIST_I2D (pl_2d, XGL_PT_I2D, NULL, 6,
                          pts_i2d_pgon1);
    xgl_polygon (ctx, XGL_FACET_NONE, 0, 0, 1, &pl_2d);

    /*
     * set to dotted line pattern object, set the line color
     * to cyan (bits in the pattern with the value of 1
     * will be CYAN), set line style to alternate line pattern,
     * set the line alternate color to green (bits in the
     * pattern with a value of 0 will be BLUE),
     * and then draw the line
     */
    ln_fg_color = cyan_color;
    ln_bg_color = blue_color;
    pgon_front_color = white_color;
```

```
xgl_object_set (ctx,
              XGL_CTX_EDGE_PATTERN, xgl_lpat_dotted,
              XGL_CTX_EDGE_STYLE, XGL_LINE_ALT_PATTERNED,
              XGL_CTX_EDGE_COLOR, &ln_fg_color,
              XGL_CTX_EDGE_ALT_COLOR, &ln_bg_color,
              XGL_CTX_SURF_FRONT_COLOR, &pgon_front_color,
              NULL);

pl_2d.pts.i2d = pts_i2d_pgon2;
xgl_polygon (ctx, XGL_FACET_NONE, 0, 0, 1, &pl_2d);

/*
 * set to Xgl dashed-dotted line pattern object,
 * set the line color to white, the alternate
 * line color to red, and then draw the line
 * NOTE: still doing alternate line colors
 */
ln_fg_color = white_color;
ln_bg_color = red_color;
pgon_front_color = blue_color;
xgl_object_set (ctx,
              XGL_CTX_EDGE_PATTERN, xgl_lpat_dashed_dotted,
              XGL_CTX_EDGE_COLOR, &ln_fg_color,
              XGL_CTX_EDGE_ALT_COLOR, &ln_bg_color,
              XGL_CTX_SURF_FRONT_COLOR, &pgon_front_color,
              NULL);

pl_2d.pts.i2d = pts_i2d_pgon3;
xgl_polygon (ctx, XGL_FACET_NONE, 0, 0, 1, &pl_2d);

/*
 * set our own dashed line pattern object, set the
 * line style to patterned, set the line foreground
 * pixel color ( bits in the pattern with the value
 * of 1 will be WHITE), copy the point data into
 * the Xgl data structure, and then draw the line
 */
ln_fg_color = white_color;
pgon_front_color = yellow_color;
xgl_object_set (ctx,
              XGL_CTX_EDGE_STYLE, XGL_LINE_PATTERNED,
              XGL_CTX_EDGE_COLOR, &ln_fg_color,
              XGL_CTX_SURF_FRONT_COLOR, &pgon_front_color,
              NULL);
pl_2d.pts.i2d = pts_i2d_pgon4;
```

```
    /*
     * loop along changing the offset into the line pattern and
     * draw the line
     */
    {
          Xgl_sgn32            i;
          Xgl_object           lpat;
          float                lpatdata[6];

          lpatdata[0] = lpatdata[1] = 2.;
          lpatdata[2] = lpatdata[3] = 4.;
          lpatdata[4] = lpatdata[5] = 1.;

          lpat = xgl_object_create (sys_st, XGL_LPAT, NULL,
                     XGL_LPAT_DATA_SIZE, 6,
                     XGL_LPAT_DATA, lpatdata,
                     NULL);

          xgl_object_set (ctx, XGL_CTX_EDGE_PATTERN, lpat, NULL);

          xgl_polygon (ctx, XGL_FACET_NONE, 0, 0, 1, &pl_2d);

          xgl_object_set (ctx,
                 XGL_CTX_EDGE_STYLE, XGL_LINE_SOLID,
                 NULL);

          xgl_object_destroy (lpat);
    }
}
```

Markers 13

This chapter discusses the Marker object and its associated attributes. It includes information on the following topics:

- Using the markers predefined by the XGL library
- Creating new markers
- Example programs

Introduction to the Marker Object

A marker is a two-dimensional symbol that represents a location in 2D or 3D space. For example, when rendering a weather map, markers can show the location of rain and thus provide a quick visual display of rain activity.

XGL provides a set of markers that an application can use, or the application can define its own markers using Marker objects. XGL-supplied markers are defined by the System State object and are available for application use at any time. Markers defined by the Marker object are available for rendering when the Marker object is attached to a Context object.

A marker is drawn centered at each point in a point list passed to the `xgl_multimarker()` operator. Markers can be rendered from a 2D or 3D Context; if the Context is 3D, the Marker location points are transformed, if necessary, on their way through the pipeline, but the Markers themselves are rendered parallel to the display in VDC or in DC.

If a Marker location point is outside the clipped bounds, the Marker is not drawn. If the location is inside the clipping bounds, the Marker is drawn but is clipped at the clipping bounds. A marker can be assigned color, but since the marker is not a surface, its color is not affected by lighting. The marker color is, however, affected by depth cueing when depth cueing is enabled. A marker can be scaled for rendering.

Using the Predefined Markers

XGL supplies eight predefined markers, which markers are shown in Figure 13-1. The predefined markers are Marker objects that the System State object creates when XGL is initialized. Predefined Marker objects are read-only objects that cannot be modified or destroyed.

Marker Name	Looks Like:	Marker Name	Looks Like:
xgl_marker_dot	•	xgl_marker_cross	×
xgl_marker_plus	+	xgl_marker_square	□
xgl_marker_asterisk	✳	xgl_marker_bowtie_ne	
xgl_marker_circle	○	xgl_marker_bowtie_nw	

Figure 13-1 XGL Predefined Markers

To use a predefined Marker, the application sets the Context attribute `XGL_CTX_MARKER` to the marker name, as in the following example:

```
xgl_object_set(ctx, XGL_CTX_MARKER, xgl_marker_dot, NULL);
```

The default Marker is `xgl_marker_plus`.

Creating a New Marker

Applications can create their own markers using Marker objects. A Marker object is created with the `xgl_object_create()` operator, using `XGL_MARKER` as the *type* parameter value.

```
marker = xgl_object_create(sys_st, XGL_MARKER, NULL, NULL);
```

Marker objects consist of a list of 2D coordinates that define one or more polylines that comprise the marker. Marker object descriptions are limited to 128 float 2D coordinates; the point type can be *Xgl_pt_f2d*, *Xgl_pt_color_f2d*, or *Xgl_pt_flag_f2d*, although color and flag data in the point type is ignored if supplied. The coordinates can be any float values but are normally between -0.5 and 0.5 in both the *x* and *y* directions.

When creating a Marker description, the application fills in an *Xgl_pt_list_list* data structure and sets the Marker object description to this data structure with the Marker attribute `XGL_MARKER_DESCRIPTION`, as shown in thefollowing code fragment.

```
new_marker = xgl_object_create(sys_st, XGL_MARKER, NULL,
                  XGL_MARKER_DESCRIPTION, &marker_pt_list_list,
                  NULL);
```

The application then attaches the new Marker to a Context with the `XGL_CTX_MARKER` attribute.

```
xgl_object_set(ctx, XGL_CTX_MARKER, new_marker, NULL);
```

Note that if the application needs to get a Marker description, the application need only declare the *Xgl_pt_list_list* structure and pass XGL a pointer to it. XGL assumes that the application will not know the size of the point lists and thus will not be able to allocate memory to hold the point list information; therefore, XGL allocates the memory for the *Xgl_pt_list_list* structure on behalf of the application. Once the memory is allocated, however, it is the responsibility of the application to free the memory with a free system call. The following lines of code show how to free the memory for an *Xgl_pt_list_list* data structure:

```
free (point_list_list.pt_lists->pts.f2d);
free (point_list_list.pt_lists);
```

Note – For information on markers for the XGL-provided CGM device, see "CGM Device Object" on page 66.

Context Marker Attributes

The attribute `XGL_CTX_MARKER` defines the Marker symbol that is drawn by `xgl_multimarker()` for a Context at each point in the point list passed to `xgl_multimarker()`.

The size of a Marker is controlled by the `XGL_CTX_MARKER_SCALE_FACTOR` attribute. This attribute specifies a scale factor by which the nominal Marker width of the Device (one pixel for Raster devices) is multiplied to produce the Marker width in device coordinates. The default value of `XGL_CTX_MARKER_SCALE_FACTOR` is 1.0. A scale factor must be provided for all predefined markers but the dot marker, or the specified marker will be rendered as one pixel. The predefined dot marker is not scaled by `XGL_CTX_MARKER_SCALE_FACTOR`. The dot marker always draws a 1-pixel dot.

Markers are always drawn parallel to the display. In 3D Contexts, the size of Markers is not affected by perspective transformations; in other words, Markers at different z values are the same size.

Each marker is a single color. Markers in a Context can all have the same color, or each marker's color can be specified individually. The attribute `XGL_CTX_MARKER_COLOR` sets the color of all Markers within a Context. If the point data supplied to the `xgl_multimarker()` primitive has color data for individual points (point type *Xgl_pt_color_f2d*), the attribute `XGL_CTX_MARKER_COLOR_SELECTOR` can specify that the color used for the Marker is taken from the point data. `XGL_CTX_MARKER_COLOR_SELECTOR` has the following values:

XGL_MARKER_COLOR_CONTEXT
Marker color is rendered as defined by XGL_CTX_MARKER_COLOR. Depth cueing affects the final appearance of a marker in 3D.

XGL_MARKER_COLOR_POINT
Marker color is taken from the point data if the point data includes colors for individual points. If the primitive data does not specify colors for individual points, the color defined by XGL_CTX_MARKER_COLOR is used.

The default value for XGL_CTX_MARKER_COLOR_SELECTOR is XGL_MARKER_COLOR_POINT.

The attributes XGL_CTX_MARKER_AA_BLEND_EQ, XGL_CTX_MARKER_AA_FILTER_WIDTH, and XGL_CTX_MARKER_FILTER_SHAPE control whether antialiasing is performed and define how markers are antialiased. For information on antialiasing, see the *XGL Reference Manual.*

Marker Example Programs

The following programs illustrate the use of predefined Markers and show how an application can define its own Marker. Each example is a fragment of a larger program. The complete program includes ex_utils.c and prims_2d_main.c, both of which are listed in Appendix B, as well as all the examples listed in this section and the primitive examples listed in Chapter 7, "Primitives and Graphics Context Attributes". To compile the complete program, type make prims_2d in the example program directory. The compiled program allows you to look at all the primitive example programs.

Predefined Marker Example Program

The program prims_2d_marker.c illustrates the predefined Markers. Figure 13-2 on page 378 shows the program output.

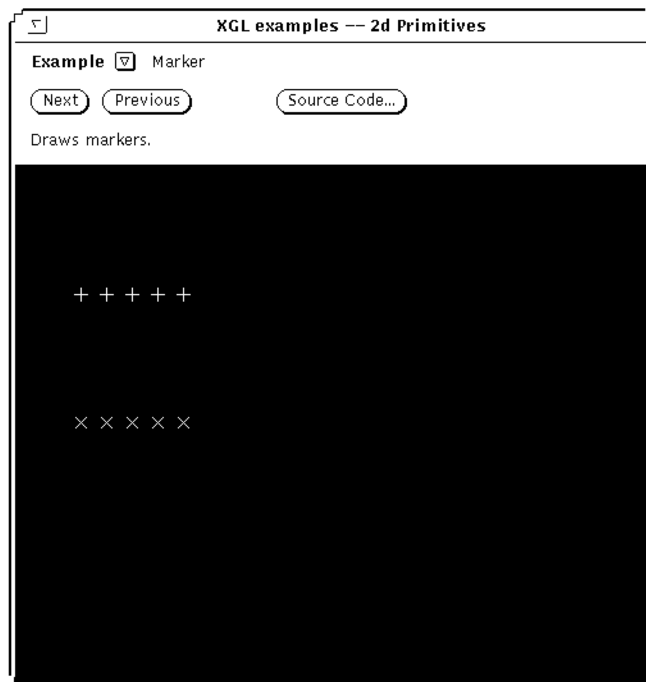


Figure 13-2 Output of `prims_2d_marker.c`

Code Example 13-1 Predefined Marker Example

```
/*
* prims_2d_marker.c
*/
#include "ex.h"

prims_2d_marker (Xgl_object   ctx)
{
    Xgl_pt_i2d          pts[20];
    Xgl_pt_list         pl[1];
    Xgl_color           color;

    xgl_object_set (ctx,
                  XGL_CTX_MARKER_SCALE_FACTOR, 10.0,
                  NULL);

    /* set up the Xgl_pt_list structure */
    pl[0].pt_type = XGL_PT_I2D;
```

```
    pl[0].bbox = NULL;
    pl[0].num_pts = 5;
    pl[0].pts.i2d = pts;

    /* draw markers in white as plus signs */

    color = white_color;
    xgl_object_set (ctx,
                  XGL_CTX_MARKER_COLOR, &color,
                  XGL_CTX_MARKER, xgl_marker_plus,
                  NULL);

    pts[0].x = 50;
    pts[0].y = 100;
    pts[1].x = 70;
    pts[1].y = 100;
    pts[2].x = 90;
    pts[2].y = 100;
    pts[3].x = 110;
    pts[3].y = 100;
    pts[4].x = 130;
    pts[4].y = 100;

    xgl_multimarker (ctx, pl);

    /* draw markers in magenta as crosses */
    color = magenta_color;
    xgl_object_set (ctx,
                  XGL_CTX_MARKER_COLOR, &color,
                  XGL_CTX_MARKER, xgl_marker_cross,
                  NULL);

    pts[0].x = 50;
    pts[0].y = 200;
    pts[1].x = 70;
    pts[1].y = 200;
    pts[2].x = 90;
    pts[2].y = 200;
    pts[3].x = 110;
    pts[3].y = 200;
    pts[4].x = 130;
    pts[4].y = 200;

    xgl_multimarker (ctx, pl);
}
```

User-defined Marker Example Program

The program `prims_2d_umarker.c` illustrates user-defined markers. Figure 13-3 shows the output of the program.

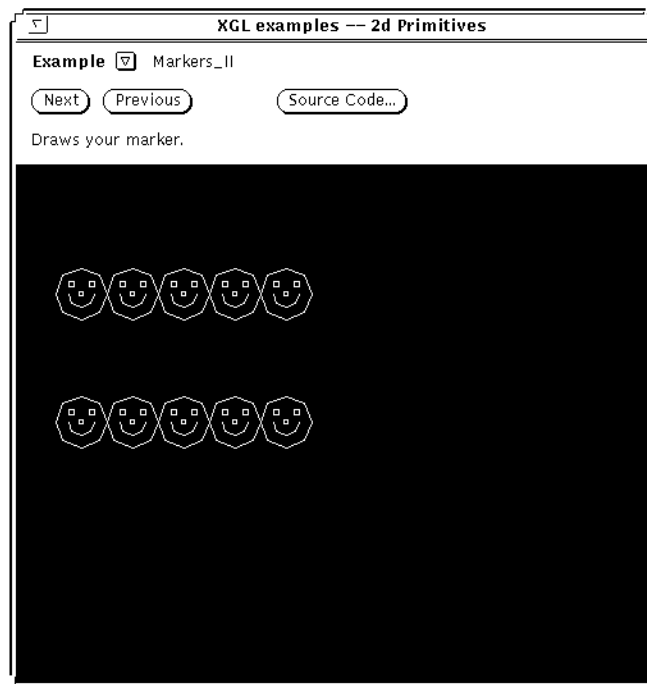


Figure 13-3 Output of `prims_2d_umarker.c`

Code Example 13-2 User-defined Marker Example

```
/*
 * prims_2d_umarker.c
 */

#include "ex.h"

#define    XGL_SET_PT_LIST(arg, argtype, argbb, argcnt, PTS_TYPE,
argpts) do { \
              (arg).pt_type = (argtype);\
              (arg).bbox = (argbb);\
              (arg).num_pts = (argcnt);\
```

```
          (arg).pts.PTS_TYPE = (argpts);\
          } while (0)

/* data for user defined markers */
static    Xgl_pt_f2d     head_circle[9] = {{0.0, -0.5},
                              {0.353553391, -0.353553391},
                           {0.5, 0.0},
                           {0.353553391, 0.353553391},
                           {0.0, 0.5},
                           {-0.353553391, 0.353553391},
                           {-0.5, 0.0},
                           {-0.353553391, -0.353553391},
                           {0.0, -0.5}};
static    Xgl_pt_f2d     mouth_circle[5] = {{0.5/2., 0.0},
                            {0.353553391/2., 0.353553391/2.},
                            {0.0, 0.5/2.},
                            {-0.353553391/2., 0.353553391/2.},
                            {-0.5/2., 0.0}};
static    Xgl_pt_f2d     nose_square[5] = {{-0.03,-0.03},
                           { 0.03,-0.03},
                           { 0.03, 0.03},
                           {-0.03, 0.03},
                           {-0.03,-0.03}};
static    Xgl_pt_f2d     eye_square[5] = {{-0.25,-0.25},
                           {-0.15,-0.25},
                           {-0.15,-0.15},
                           {-0.25,-0.15},
                           {-0.25,-0.25}};
static    Xgl_pt_f2d     eye_square2[5] = {{ 0.25,-0.25},
                           { 0.15,-0.25},
                           { 0.15,-0.15},
                           { 0.25,-0.15},
                           { 0.25,-0.25}};
static    Xgl_pt_list        face_pt_list[5];
static    Xgl_pt_list_list   face_pt_list_list = {0, 5,
                                  &face_pt_list[0]};

prims_2d_umarker (Xgl_object  ctx)

    Xgl_pt_i2d        pts[20];
    Xgl_pt_list       pl[1];
    Xgl_color         color;
    Xgl_Object        face_marker;

    /* create our own marker (the smiley-face marker) and assign
     * its data */
```

```
face_marker = xgl_object_create(sys_st, XGL_MARKER, NULL, 0);

XGL_SET_PT_LIST(face_pt_list[0], XGL_PT_F2D,NULL, 9, f2d,
               &head_circle[0]);
XGL_SET_PT_LIST(face_pt_list[1], XGL_PT_F2D,NULL, 5, f2d,
               &mouth_circle[0]);
XGL_SET_PT_LIST(face_pt_list[2], XGL_PT_F2D,NULL, 5, f2d,
               &eye_square[0]);
XGL_SET_PT_LIST(face_pt_list[3], XGL_PT_F2D,NULL, 5, f2d,
               &eye_square2[0]);
XGL_SET_PT_LIST(face_pt_list[4], XGL_PT_F2D,NULL, 5, f2d,
               &nose_square[0]);

xgl_object_set(face_marker,
          XGL_MARKER_DESCRIPTION, &face_pt_list_list,
          NULL);

xgl_object_set (ctx,
              XGL_CTX_MARKER_SCALE_FACTOR, 40.0,
              NULL);

/* setup the Xgl_pt_list structure */

pl[0].pt_type = XGL_PT_I2D;
pl[0].bbox    = NULL;
pl[0].num_pts = 5;
pl[0].pts.i2d = pts;

/* draw row of smiley-face markers in white */

color = white_color;
xgl_object_set (ctx,
              XGL_CTX_MARKER_COLOR, &color,
              XGL_CTX_MARKER, face_marker,
              NULL);

pts[0].x = 50;
pts[0].y = 100;
pts[1].x = 90;
pts[1].y = 100;
pts[2].x = 130;
pts[2].y = 100;
pts[3].x = 170;
pts[3].y = 100;
pts[4].x = 210;
```

```
    pts[4].y = 100;

    xgl_multimarker (ctx, pl);

    /* draw row of smiley-face markers in magenta */

    color = magenta_color;
    xgl_object_set (ctx,
                  XGL_CTX_MARKER_COLOR, &color,
                  NULL);

    pts[0].x = 50;
    pts[0].y = 200;
    pts[1].x = 90;
    pts[1].y = 200;
    pts[2].x = 130;
    pts[2].y = 200;
    pts[3].x = 170;
    pts[3].y = 200;
    pts[4].x = 210;
    pts[4].y = 200;

    xgl_multimarker (ctx, pl);

    /* destroy the marker we created and reset to default dot
     * marker
     */

    xgl_object_destroy(face_marker);
    xgl_object_set (ctx,
                  XGL_CTX_MARKER, xgl_marker_dot,
                  NULL);
}
```

Picking 14

This chapter provides information on XGL picking operators and attributes. It includes an example program illustrating picking.

Introduction to XGL Picking

In interactive graphics applications, the user may need to select graphics primitives from the ones visible on the display. This is often done with the aid of a pointing device (such as a mouse).

Once a picking region is defined and picking is enabled, any primitive that is sent into the XGL pipeline is checked against the picking region. This comparison is done in screen coordinates after the primitives have had their coordinate values transformed. In general, if any part of the transformed primitive is found to be in the pick region, that primitive is picked. The parts of a polygon primitive that can be picked depend on the XGL_CTX_PICK_SURF_STYLE attribute. When a primitive is picked, the pick buffer is updated with the pick identifier information in the Context at that time.

Every Context has a *pick buffer* that stores information about pick events. A *pick event* occurs whenever a part of a primitive that is sent down the pipeline after picking is enabled falls within the *pick aperture*. The pick aperture is the area (for a 2D Context) or volume (for a 3D Context) used to test primitives for picking. If a primitive falls within the bounds set by the attribute XGL_CTX_PICK_APERTURE (set in Device Coordinates), the entire primitive is picked when any part of the primitive is picked.

A 3D Context requires a range of z values as well as x and y values in order to define a picking volume.

Note – The XGL viewing transform can scale and translate the z values of a primitive to unexpectedly high values. Be sure to set the maximum z value appropriately. If in doubt, or if you just want to pick in x and y, call `xgl_object_get()`, inquiring on `XGL_DEV_MAXIMUM_COORDINATES`, and use the z value as the maximum z picking range (use 0 as the minimum).

The *pick identifier* attributes `XGL_CTX_PICK_ID1` and `XGL_CTX_PICK_ID2` are added as entries to the pick buffer according to certain rules. The application should set the pick identifier attributes to unique values as each primitive is sent down the pipeline. That way, the application can tell which primitive was picked. Only one pick aperture can be active at any one time.

If Z-buffering is enabled during a pick, primitives are checked for visibility against the current Z-buffer values. If the primitive is obscured within the picking region (using the Z-buffer hidden surface algorithm), the object will not be picked even if it lands in the pick region.

Picking by itself does not update the Z-buffer, only rendering does. If the Z-buffer has not been properly updated (by re-rendering the image with Z-buffering enabled whenever a change occurs in the image), the Z-buffered picking values will most likely be incorrect. The application should ensure that the Z-buffer state reflects the primitives it is trying to pick.

Picking Attributes

The attribute `XGL_CTX_PICK_ENABLE` must be `TRUE` for picking to occur. All rendering primitives except `xgl_image()` can be picked. `XGL_CTX_PICK_STYLE` can only be changed when picking is disabled. The size of the pick buffer is determined by the attribute `XGL_CTX_PICK_BUFFER_SIZE`. The default size of the pick buffer is 256. It can be changed to any nonnegative value.

The attributes `XGL_CTX_PICK_ID_1` and `XGL_CTX_PICK_ID_2` identify the primitives that were picked. Having two pick IDs enables the application to associate more information with a picked primitive. The application can identify picked primitives, depending on how the pick IDs were set before the

primitives were sent through the pipeline. The current pick ID must differ from the last entry in the buffer, or it will not be stored again. The default value for the pick ID attributes is 0.

The attribute XGL_CTX_PICK_STYLE defines what happens when pick IDs are added to a full pick buffer. It also controls how pick events are returned to the application when the operator xgl_pick_get_identifiers() is invoked. The two pick styles are:

XGL_PICK_FIRST_N
The first *n* pick events (*n* being the pick buffer size) are stored in the buffer. Once the buffer is full, later pick events are ignored. The events are returned in the order they were stored, the first event being the first one returned.

XGL_PICK_LAST_N
The pick events are stored in a first-in-first-out (FIFO) queue. When the pick buffer is full, the oldest event is removed to make room for the most recent event. The events are returned in reverse order to that in which they were stored; the most recent event in the buffer is the first one returned.

The attribute XGL_CTX_PICK_SURF_STYLE determines which parts of the surface primitives are pickable:

XGL_PICK_SURF_AS_SOLID: The entire surface is pickable.

XGL_PICK_SURF_AS_FILL_STYLE: The pickable part is device-dependent if the polygon is stipple-filled. (A stipple-filled polygon will either be picked like a solid-filled one, or only be pickable on the visible parts of the fill pattern.) Empty polygons are not pickable, unless their edges are made explicitly visible with the vertex visibility flag. Then only the visible sections of the edges are pickable. Hollow polygons can be picked only at their edges.

XGL Picking Operators

The picking operators are `xgl_pick_clear()` and `xgl_pick_get_identifiers()`.

Clearing the Pick Buffer

The operator `xgl_pick_clear()` clears the pick buffer associated with the Context `ctx` to the *nothing-picked* state.

```
void xgl_pick_clear (Xgl_ctxctx);
```

Identifying Picked Primitives

The `xgl_pick_get_identifiers()` operator returns the identifiers (IDs) of picked primitives stored in the Context `ctx`. The returned pick IDs are stored in the array `pick_id_list`, which is passed as an argument to the operator. The application must have allocated `pick_id_list` to hold as many *Xgl_pick_info* structures as are in the Context pick buffer. The number of structures in the Context pick buffer is set by the attribute `XGL_CTX_PICK_BUFFER_SIZE`. The default value is 256. If the pick buffer overflows, the oldest pick information may be dropped, or the most recent information may be ignored, depending on the value of `XGL_CTX_PICK_STYLE`. The order in which the pick events are returned depends on the value of the attribute `XGL_CTX_PICK_STYLE`.

```
void xgl_pick_get_identifiers(
    Xgl_ctx           ctx,
    Xgl_usgn32        *count,
    Xgl_pick_info     pick_id_list[] );
```

Picking Example

The example program `pick_2d_prims.c` draws white plus-sign markers, a blue polygon, and a yellow polyline that can be picked interactively using the right mouse button. This program uses the examples utilities package `ex.utils.c` to pick when the left mouse button is pressed. The application must allocate enough space to hold all possible return *Xgl_pick_info* structures. If anything is picked, information on picked geometry will be returned to the application program. For example, if the polygon is selected, the statement "Polygon picked" is displayed. If nothing is selected, the statement "Nothing picked" is displayed.

To compile this program, type `make pick_2d` in the example program directory. The compiled program includes `ex_utils.c` and `pick_2d_main.c`, which are listed in Appendix B. Figure 14-1 shows the output of the program.

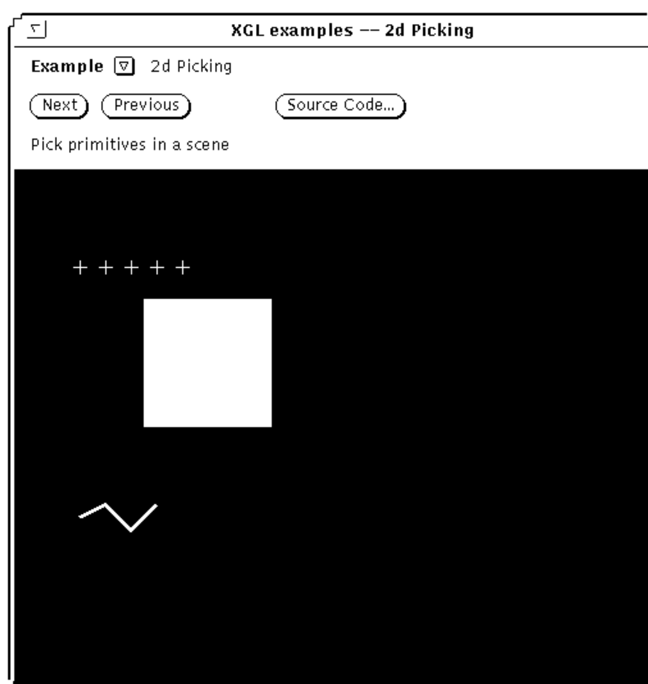


Figure 14-1 Output of `pick_2d_prims.c`

Code Example 14-1 Picking Example

```
/*
 * pick_2d_prims.c
 */
#include "ex.h"
#include <xgl/xgl.h>
#include <malloc.h>

extern Xgl_color_typeex_color_type;

/* Pick Identifiers for the primitives */

#define PICK_ID_MARKER1
#define PICK_ID_PLINE2
#define PICK_ID_PGON3

/* XGL provides 2 pick identifiers.  For this example, we are only
 * using one of them: XGL_CTX_PICK_ID_1.  We draw each primitive in
 * a different color.  We set all the colors at once, before the
 * primitives are drawn.
 *
 * The pick identifier is set just before each primitive is drawn
 * because the pick identifier applies to _all_ subsequent
 * primitives.To uniquely
 * identify each primitive, we need to change the pick id for it.
 *
 * A performance tip: if you are drawing lots of primitives, don't
set
 * the pick identifier if you are not picking; the "picking" variable
 * is used for this.
 */

int         picking = TRUE; /* This is TRUE if we ar picking.*/

pick_draw_scene (Xgl_object     ctx)

    Xgl_pt_i2d          pts[20];
    Xgl_pt_list         pl[1];
    Xgl_color           marker_color;
    Xgl_color           pgon_color;
    Xgl_color           pline_color;

    /* draw markers in white as plus signs */
    marker_color = white_color;
```

```
/* draw square polygon in cyan */
pgon_color = cyan_color;
/* draw 3-segment polyline in yellow */
pline_color = yellow_color;
xgl_object_set (ctx,
          XGL_CTX_MARKER_SCALE_FACTOR, 10.0,
          XGL_CTX_MARKER_COLOR, &marker_color,
          XGL_CTX_MARKER, xgl_marker_plus,
          XGL_CTX_SURF_FRONT_COLOR, &pgon_color,
          XGL_CTX_LINE_COLOR, &pline_color,
          XGL_CTX_LINE_WIDTH_SCALE_FACTOR, 3.0,
          NULL);

/* setup the Xgl_pt_list structure */

pl[0].pt_type = XGL_PT_I2D;
pl[0].bbox = NULL;
pl[0].pts.i2d = pts;

pl[0].num_pts = 5;
pts[0].x = 50;
pts[0].y = 75;
pts[1].x = 70;
pts[1].y = 75;
pts[2].x = 90;
pts[2].y = 75;
pts[3].x = 110;
pts[3].y = 75;
pts[4].x = 130;
pts[4].y = 75;

if (picking)
   xgl_object_set (ctx, XGL_CTX_PICK_ID_1, PICK_ID_MARKER,
                   NULL);
xgl_multimarker (ctx, pl);

/* Draw a square as a polygon */

pl[0].num_pts = 4;

pts[0].x = 100;
pts[0].y = 100;
pts[1].x = 200;
pts[1].y = 100;
pts[2].x = 200;
pts[2].y = 200;
```

```
    pts[3].x = 100;
    pts[3].y = 200;

    if (picking)
       xgl_object_set (ctx, XGL_CTX_PICK_ID_1, PICK_ID_PGON,
                        NULL);
    xgl_polygon (ctx, XGL_FACET_NONE, NULL, NULL, 1, pl);

    /* Draw a polyline */

    pl[0].num_pts = 4;

    pts[0].x = 50;
    pts[0].y = 270;
    pts[1].x = 70;
    pts[1].y = 260;
    pts[2].x = 90;
    pts[2].y = 280;
    pts[3].x = 110;
    pts[3].y = 260;
    pts[4].x = 130;
    pts[4].y = 270;

    if (picking)
       xgl_object_set (ctx, XGL_CTX_PICK_ID_1, PICK_ID_PLINE,
                        NULL);
    xgl_multipolyline (ctx, NULL, 1, pl);
}

/* For this example, the pick aperture is a square
 * (2 * APERTURE_SIZE) pixels
 * in width and height centered at the event X, Y.
 */

#define APERTURE_SIZE  5

/* The example utilities package (ex_utils.c) calls this function
 * when the left (ACTION_SELECT) mouse button goes down.
 */

pick_do_pick (
    Xgl_object          ctx,
    int                 x,
    int                 y)
{
    Xgl_bounds_f2d      pick_aperture;
```

```
Xgl_usgn32           count;
int               i;
static Xgl_pick_info *id_list = NULL;

pick_aperture.xmin = x - APERTURE_SIZE;
pick_aperture.xmax = x + APERTURE_SIZE;
pick_aperture.ymin = y - APERTURE_SIZE;
pick_aperture.ymax = y + APERTURE_SIZE;

/* Set the pick aperture before turning on picking */

xgl_object_set (ctx,
          XGL_CTX_PICK_APERTURE, &pick_aperture,
          XGL_CTX_PICK_ENABLE, TRUE,
          NULL);
xgl_pick_clear (ctx);
picking = TRUE;

/*
 * "Draw" the scene. Note: nothing is really drawn on the
 * screen because we have turned XGL_CTX_PICK_ENABLE on,
 * which turns rendering off.
 * The primitives are just processed for picking.
 */
pick_draw_scene (ctx);

picking = FALSE;
xgl_object_set (ctx,
          XGL_CTX_PICK_ENABLE, FALSE,
          NULL);

/* See what, if anything, we have picked */

/*
 * But, first, we need to allocate enough space to hold all the
 * possible returned pick_info structures.
 */
if (id_list == NULL) {
   Xgl_usgn32          buff_size;

   xgl_object_get (ctx, XGL_CTX_PICK_BUFFER_SIZE,
                   &buff_size);
   id_list = (Xgl_pick_info *) malloc (sizeof (Xgl_pick_info) *
                 buff_size);
   if (!id_list) {
       fprintf (stderr, "pick_2d: malloc failed!\n");
```

```
                exit (1);
            }
        }
        xgl_pick_get_identifiers (ctx, &count, id_list);

        if (count == 0)
            printf ("Nothing picked.\n");
        else {
            for (i = 0; i < count; i++) {
                switch (id_list[i].id1) {
                case PICK_ID_MARKER:
                    printf ("Markers picked.\n");
                    break;
                case PICK_ID_PGON:
                    printf ("Polygon picked.\n");
                    break;
                case PICK_ID_PLINE:
                    printf ("Polyline picked.\n");
                    break;
                default:
                    printf ("Invalid pick_id.\n");
                break;
                }
            }
        }
}
```

Lighting, Shading, and Depth Cueing

This chapter discusses the XGL lighting model. It includes information on the following topics:

- Available light types
- Reflection components
- Equations that govern the XGL light reflection model
- Information on using XGL for lighting and depth cueing
- Example program of lighting in indexed color space

Introduction to the 3D Rendering Pipeline

The XGL 3D rendering pipeline draws geometric primitives with varying degrees of realism. The 3D rendering pipeline consists of several stages: *lighting, shading, depth cueing,* and *color mapping*. The application can configure XGL for all or just some of the operations, depending on the final visual result desired and the time for producing an image.

Lighting

The lighting stage is early in the rendering pipeline. It applies only to surfaces defined in 3D. The current release of XGL limits lighting to surfaces composed of polygonal facets.

Lighting simulates the physical interaction of light reflecting from surfaces. The color and intensity of reflected light depends on the properties of lights and surfaces, and on the orientation of a surface relative to lights and the observer. The lighting calculations in XGL are executed on each polygonal facet independent of all other facets; effects such as shadows and reflections that are achievable with ray tracing are not provided.

XGL performs lighting calculations in RGB and indexed color spaces. Some conversion of colors may be necessary, for example, if the frame buffer operates in a color space that is different from that of the lighting calculation.

Lighting calculations span the major aspects of XGL including the geometric and color characteristics of a surface primitive, the rendering attributes of a 3D Context, and the objects associated with 3D Contexts, such as Lights, Transforms, and the Device's Color Map.

Shading

Shading computations are closely linked to lighting computations. The shading method determines how the rendering parameters are interpolated across a facet and where lighting calculations are performed. Lighting may precede or follow shading. Two common shading methods are:

- Color shading, which calculates the light reflected from the vertices of a facet, and then interpolates these reflected colors across the facet.
- Normal shading, which interpolates reflectance normal vectors (specified at vertices) across the facet, and then calculates the light reflected from intermediate locations on the facet that correspond to pixel positions. (The current release of XGL does not support normal shading.)

Depth Cueing

Depth cueing follows lighting and shading. It enhances depth perception by blending colors in the image. XGL blends colors by modulating the colors calculated in the preceding stage by another specified color. The amount of blending depends on the distance of a primitive from the observer. Depth cueing applies to all primitives when it is enabled, not just surfaces.

Surface Primitives

Polygonal surface primitives consist of ordered lists of vertices that define facets. Each vertex possesses a 3D position in Model Coordinates and possibly a reflectance unit normal vector, a color, and flag information (usually for specifying edge highlighting). A surface primitive may also contain a normal vector or color, or both, for each polygonal facet. Lighting calculations use this information in different ways, depending on the desired visual appearance.

The 3D Context in Lighting

A 3D Context has attributes that control the global rendering properties of materials, such as material color (used when a surface primitive does not have a color bound to it) and the reflection coefficients for the various components of reflected light.

A 3D Context also contains objects that affect lighting. The lights themselves can be positioned and aimed to illuminate graphical objects for a desired effect. The transformation pipeline implicitly gives the position of the observer, and the Model Transform maps the geometric data to World Coordinates where lighting takes place. Finally, the Context renders into a Device with a Color Map, which determines how to interpret, calculate, and map material colors into reflected colors.

Basic Lighting and Shading Concepts

This section describes basic concepts of the XGL lighting model for 3D polygonal surfaces. It discusses the factors involved in creating graphic realism in the XGL 3D rendering pipeline.

Colors

Three types of color make up the lighting portion of the 3D rendering pipeline:

- **Material Color**: Intrinsic to the material that geometric surfaces represent. For example, a red plastic pen is intrinsically red, even in a dark room.
- **Light Color**: The color of a light source. For example, a white light with a green filter emits a narrow range of wavelengths in the green part of the spectrum.

- **Reflected Color**: The characterization of the color of light reflected from a surface. Reflected color depends on the material color and the properties of the light sources including colors, directions, and positions.

Reflection Components

In the XGL lighting model, the light reflected from a surface consists of basic components with unique properties. These are called *reflection components*. This model is partly physical and partly empirical. It follows the lighting model outlined by PHIGS PLUS, which is a library of extensions to the PHIGS standard library. This section describes reflection components without equations. "Lighting Equations" on page 400 gives a quantitative description.

- **Ambient Reflection**: The ambient reflection component adds a uniform brightness to the light reflected from surfaces, simulating light reflected from multiple surfaces. Ambient light does not come from any single direction or position but comes fom all directions with uniform intensity. Ambient reflections do not depend on the orientation or position of a surface. As the name implies, ambient light provides low-level background illumination.
- **Diffuse Reflection**: The diffuse reflection component models the effect of directional light being scattered in all directions after reflection from a rough matte-like surface. The intensity of reflected light depends on the incident angle of the incoming light rays and the reflection coefficient of the surface. The diffuse reflection falls off with the cosine of the angle between the surface normal vector and the direction vector pointing to the light source, a property known as Lambert's cosine law.
- **Specular Reflection**: The specular reflection component produces highlights or glints on glossy or shiny surfaces. This type of reflection models the law of reflection in geometrical optics, where the reflectance angle equals the incident angle. The amount of specular reflection from a point on a surface depends on the direction of the viewer from that point. If that direction closely coincides with the reflectance direction, the viewer perceives a bright spot at that point on the surface.

Light Sources

XGL has four types of light sources, each having different properties for simple to complex illumination requirements. An application can combine any number and type of light sources to produce a desired effect. For a simple illustration of XGL light sources, see Figure 1-10 on page 12.

- **Ambient Light**: An *ambient* light has only an ambient reflection component. It does not have diffuse and specular reflection components.
- **Directional Light**: A *directional* light has parallel light rays. Direct sunlight is a physical example of this type of light. A directional light contributes to diffuse and specular reflections, which in turn depend on the orientation of a surface, but not its position. A directional light does not contribute to ambient reflections.
- **Positional Light**: A *positional* light radiates light rays uniformly from a single point in all directions. (A bare incandescent light bulb radiates light in this manner.) A positional light contributes to diffuse and specular reflections, which in turn depend on the orientation and position of a surface. A positional light does not contribute to ambient reflections. The intensity profile of a positional light depends on the distance of the light from the surface.
- **Spot Light**: A *spot* light radiates light rays from a single point. A spot light has both a position and a direction. The ray with the maximum intensity propagates in this direction. Rays lying off this principal axis decrease in intensity with the angle away from the axis. The light from a spot light affects surfaces lying in a cone. The intensity profile of a spot light depends on the distance of the light from the surface. A spot light contributes to diffuse and specular reflections, which in turn depend on the orientation and position of a surface.

Illumination and Shading

Common shading techniques are known as flat, Gouraud (color), and Phong (normal). Shading can change the appearance of polygonal facets through interpolation of color values across polygons during the scan conversion stage of rendering. Color and normal shading make polygonal representations look like smooth curved surfaces when the polygons are sufficiently small.

XGL enables applications to select an illumination type to specify where the lighting calculations take place. Depending on the illumination type and on the lights that are *on* at that time (as specified by the attribute XGL_3D_CTX_LIGHT_SWITCHES), XGL will automatically calculate surface color and shading. The current release of XGL supports four illumination types:

- **None**: No illumination with no interpolation renders polygons drawn in the material or intrinsic color.
- **None with color interpolation**: No illumination calculations are done; XGL directly interpolates vertex colors across the surface.
- **Facet**: Illumination per facet results in one lighting calculation for each polygon. XGL draws the entire polygon in this reflected color. This is known as *flat shading*. Note that a light must be turned on for facet illumination to occur, and if the Raster is indexed, a color map must be available. The vertex chosen for the lighting calculation is device-dependent.
- **Vertex**: Illumination per vertex leads to calculation of the reflected light at each vertex of each polygon. XGL draws the polygon by linearly interpolating the reflected colors at the vertices across the edges and subsequently across scan lines. This is known as *color shading*. Color shading can give the appearance of smooth surfaces. However, applications must supply unit-length reflectance normal vectors at each vertex. A vertex normal points approximately in the direction of the average of the facet normals of all facets incident on the vertex. Note that a light must be turned on for vertex illumination to occur, and if the Raster is indexed, a color map must be available.

Lighting Equations

XGL performs lighting calculations in the same color space as the destination Raster's color type. As discussed in Chapter 5, "Color," XGL supports two types of color: RGB and indexed. This section describes lighting calculations with variables and equations for these two color spaces.

The lighting equations involve variables that come from various parts of XGL. They may be stored in the 3D Context object, the Context's Light objects, the Context's Device's Color Map object, or the geometry of the primitive being rendered. Table 15-1 on page 401 describes the symbols used in the lighting calculations, where for the data types:

F = floating point
G = gray color type
I = indexed color type
RGB = RGB color type

and for the sources:
[1] = Light object
[2] = calculated
[3] = explicit or derived from object geometry
[4] = vertex color, facet color, or 3D Context object
[5] = 3D Context object
[6] = calculated from the equations that follow

Table 15-1 Lighting Variables

Symbol	Description	RGB	Indexed	Source
$\vec{L}_d$	Light source direction	3×F	3×F	[1]
L_c	Light source color	RGB	G	[1]
$\vec{L}_p$	Light source position	3×F	3×F	[1]
L_e	Light source concentration exponent	F	F	[1]
C_1, C_2	Attenuation coefficients	F	F	[1]
A_s	Spread angle	F	F	[1]
L_a	Light attenuation	F	F	[6]
$\vec{O}_p$	Object position	3×F	3×F	[3]
O_d	Object diffuse color	RGB	I	[4]
O_s	Object specular color	RGB	I	[5]
O_e	Object specular exponent	F	F	[5]
K_a	Ambient reflection coefficient	F	F	[5]
K_d	Diffuse reflection coefficient	F	F	[5]
K_s	Specular reflection coefficient	F	F	[5]
$\vec{V}_e$	Unit vector from object to eye point	3×F	3×F	[6]
$\vec{V}_r$	Unit reflection vector from object	3×F	3×F	[6]

Table 15-1 Lighting Variables *(Continued)*

Symbol	Description	RGB	Indexed	Source
$\vec{V}_l$	Unit vector from object to light source	3×F	3×F	[2]
$\vec{V}_n$	Unit normal vector to object	3×F	3×F	[6]
C_a	Ambient contribution from light source	RGB	I	[6]
C_d	Diffuse contribution from light source	RGB	I	[6]
C_s	Specular contribution from light source	RGB	I	[6]

XGL calculates the reflected light at a point on a primitive by adding the contributions from each reflection component created by active light sources (totaling N in number) as follows:

$$C = \sum_{i=1}^{N} \left(C_{a_i} + C_{d_i} + C_{s_i} \right)$$

For ambient light sources, the reflectance components are as follows:

$$C_a = K_a O_d L_c$$
$$C_d = 0$$
$$C_s = 0$$

For directional light sources, the reflectance components are as follows:

$$C_a = 0$$
$$C_d = K_d O_d L_c (\vec{V}_n \cdot -\vec{L}_d)$$
$$C_s = K_s O_s L_c (\vec{V}_e \cdot \vec{V}_r)^{O_e}$$

For positional light sources, the reflectance components are as follows:

$$C_a = 0$$
$$C_d = K_d O_d L_c (\vec{V}_n \cdot \vec{V}_l) L_a$$

$$C_s = K_s O_s L_c (\vec{V}_e \cdot \vec{V}_r)^{O_e} L_a$$

For spot light sources, the reflectance components are as follows:

$$C_a = 0$$

$$C_d = \begin{cases} K_d O_d L_c (\vec{V}_n \cdot \vec{V}_r) (-\vec{L}_d \cdot \vec{V}_l)^{L_e} L_a, & (-\vec{L}_d \cdot \vec{V}_l) \geq \cos\left(\frac{A_s}{2}\right) \\ 0 & , (-\vec{L}_d \cdot \vec{V}_l) < \cos\left(\frac{A_s}{2}\right) \end{cases}$$

$$C_s = \begin{cases} K_s O_s L_c (\vec{V}_e \cdot \vec{V}_r)^{O_e} (-\vec{L}_d \cdot \vec{V}_l)^{L_e} L_a, & (-\vec{L}_d \cdot \vec{V}_l) \geq \cos\left(\frac{A_s}{2}\right) \\ 0 & , (-\vec{L}_d \cdot \vec{V}_l) < \cos\left(\frac{A_s}{2}\right) \end{cases}$$

The light attenuation for positional and spot lights is given by:

$$L_a = 1 / \left(C_1 + C_2 \cdot norm(\vec{O}_p - \vec{L}_p) \right)$$

The light reflection vector is:

$$\vec{V}_r = 2 (\vec{V}_n \cdot \vec{V}_l) \vec{V}_n - \vec{V}_l$$

XGL replaces all negative dot product values with zero.

RGB Lighting

In RGB color space, each member of a triplet of color values stored in the frame buffer is converted to a voltage that drives the individual electron guns of a cathode ray tube. Each channel is independent of the other two.

In RGB lighting, the application specifies both material and light colors in the RGB format. XGL represents the value of each channel as a floating-point number in the range 0.0 to 1.0. If the result of a lighting calculation exceeds 1.0, XGL clamps the result to 1.0. RGB lighting applies the equations in turn to each of the three channels to produce RGB reflected colors.

RGB frame buffers require more planes of video memory than indexed frame buffers, and RGB lighting requires more computation than indexed lighting; however, RGB lighting provides more realistic images with a wider range of color variation.

XGL allows rendering in RGB even if the frame buffer is indexed. Applications do this by creating an RGB Raster on an indexed frame buffer. XGL performs the lighting calculation in RGB. Applications should have a color cube defined in the Color Map. A default color cube is created for all RGB Rasters. Then XGL dithers the colors in this color cube and writes indexes to the indexed frame buffer.

Indexed Lighting

Indexed frame buffers require significantly smaller amounts of video memory than RGB frame buffers. The indexed color method provides an indirect means of generating color on a graphics device. For each pixel, an indexed frame buffer stores an index into a lookup table containing RGB values that drive the electron guns. The palette of available colors is large, but the number of colors that can be displayed at any given moment is small.

Lighting in indexed color space has limitations because of the small choice of colors at any one time. Lighting in indexed color space is also more complicated than lighting in RGB. If an application is willing to accept these limitations, however, lighting in indexed space on indexed frame buffers is faster than lighting in RGB space and dithering the colors in a color cube.

In indexed lighting, lights are always white with varying levels of brightness. Applications specify the light intensity as a color of type *gray*. Material and reflected colors are indexes. The application partitions the color table into segments or *color ramps*, as demonstrated in the example program at the end of this chapter. (A material color index must reside in a defined color ramp. After lighting, the reflected color will reside within the same ramp.)

Color ramps typically start at black at the lowest index of the segment and increase linearly to some maximum value at the top of the ramp. For example, the color at the top could be bright saturated red. From the top, the intermediate values in the ramp decrease in brightness to black at the bottom of the ramp. Color ramps need not be linear. Both saturation and brightness may be nonlinear to give effects such as white specular highlights.

XGL calculates reflected light in indexed space in the following way. The calculation involves conversion of indexes to gray values, application of the lighting equations in gray color space, and finally conversion of the reflected gray color to an index. The process is as follows:

1. From the 3D Context, get the destination Raster's Color Map.
2. Given a color index for the diffuse material color, find the color segment in the Color Map to which the index belongs. Let the segment offset within the color table be I_s. Let the segment length be N_s.
3. Convert diffuse material color index to a floating-point gray value. The specified diffuse material color $\hat{O}_d$ is an index. Let the corresponding gray value be O_d, having a range from 0.0 to 1.0. Then,

$$O_d = \frac{\hat{O}_d - I_s}{N_s - 1}$$

4. If the specular color index is in the same segment as the diffuse color index, convert the specular color index to a floating point value. The specified specular material color $\hat{O}_s$ is an index. Let the corresponding gray value be O_s, having a range from 0.0. to 1.0. Then,

$$O_s = \frac{\hat{O}_s - I_s}{N_s - 1}$$

5. Otherwise, the specular color index does not reside in the same segment as the diffuse color index. To deal with this inconsistency, let

$$O_s = 0$$

6. Perform lighting in gray color space according to the equations given on page 400. Accumulate the reflected light contributions as a gray value C, ranging from 0.0 to 1.0
7. Clamp C so that it does not exceed 1.0.
8. Convert C to a color index residing in the original color ramp:

$$\hat{C} = truncate(C(N_s - 1) + I_s)$$

Note that in similar situations, lighting in index color space may give different results than lighting in RGB color space does. As long as application developers understand the limitations of indexed lighting, however, useful results are attainable.

XGL Lighting Attributes

The lighting calculations are affected by attributes associated with both individual Light objects and the associated 3D Context. Table 15-2 associates the parameters that affect lighting with XGL attributes.

Table 15-2 Lighting Parameters and Lighting Attributes

Symbol	XGL Lighting Attribute
$\vec{L}_d$	XGL_LIGHT_DIRECTION
L_c	XGL_LIGHT_COLOR
$\vec{L}_p$	XGL_LIGHT_POSITION
L_e	XGL_LIGHT_SPOT_EXPONENT
C_1	XGL_LIGHT_ATTENUATION_1
C_2	XGL_LIGHT_ATTENUATION_2
A_s	XGL_LIGHT_SPOT_ANGLE
O_d	XGL_CTX_SURF_FRONT_COLOR, XGL_3D_CTX_SURF_BACK_COLOR or color stored with facet or vertex
O_s	XGL_3D_CTX_SRF_FRONT_SPECULAR_COLOR XGL_3D_CTX_SURF_BACK_SPECULAR_COLOR
O_e	XGL_3D_CTX_SURF_FRONT_SPECULAR_POWER, XGL_3D_CTX_SURF_BACK_SPECULAR_POWER
K_a	XGL_3D_CTX_SURF_FRONT_AMBIANT, XGL_3D_CTX_SURF_BACK_AMBIANT
K_d	XGL_3D_CTX_SURF_FRONT_DIFFUSE, XGL_3D_CTX_SURF_BACK_DIFFUSE
K_s	XGL_3D_CTX_SURF_FRONT_SPECULAR, XGL_3D_CTX_SURF_BACK_SPECULAR

Creating a Light Object

An application can create a Light object and set its attributes to give particular lighting effects. At any time, lights can be set to any of the four light source types — ambient, directional, positional, or spot. By default, the light type is ambient. Note, however, that no Light objects are created at XGL initialization. In order to use lights, you must define them in one of two ways:

- You can creat a Light object and specify lighting attributes with `xgl_object_create()` used with the *type* parameter set to `XGL_LIGHT`. Light objects created in this way can be shared among Contexts.
- You can manipulate the array of lights defined in the 3D Context. When an application sets the number of Lights in a 3D Context, XGL creates or destroys Lights to adjust the Context's array of Lights to the specified number. XGL also adjusts the array of on/off light switches to match the number of Lights.

To define Light objects using the Context's light array, follow these steps:

1. Declare local variables for an array for the number of lights and an array for the light switches.
2. Set the 3D Context attribute `XGL_3D_CTX_LIGHT_NUM` to the number of lights.

   When this attribute is set, XGL creates new Light objects to match this attribute and changes the array of Light switches to a corresponding number. The light switches are set to `TRUE` by default. If the new number of lights exceeds the current number, XGL creates new Lights and switches and appends them to the current arrays; in this case, the Lights and switches that already existed remain unchanged. If the new number of lights is less that the current number, XGL destroys Lights and switches so that the number of lights and switches matches the attribute number.
3. Retrieve the list of lights and light switches from the Context.

   Alternatively, you can create an array of Lights using `xgl_object_create()` and attach the array to the Context.
4. Set attributes as needed. Note that the first light you define must be an ambient light.

The following code fragment illustrates the array of light objects. For additional examples, see the sample programs later in this chapter.

```
Xgl_light    light[2];
Xgl_boolean  light_switch[2];
Xgl_color    light_color;

/* Set the number of lights */
xgl_object_set (ctx, XGL_3D_CTX_LIGHT_NUM, 2,
                NULL);

/* Get lights and switches */
xgl_object_get(ctx, XGL_3D_CTX_LIGHTS, light);
xgl_object_get(ctx, XGL_3D_CTX_LIGHT_SWITCHES, light_switch);

/* Set  the values for Light 0; must be type ambient */
xgl_object_set(light[0],
                XGL_LIGHT_TYPE, XGL_LIGHT_AMBIENT,
                XGL_LIGHT_COLOR, &light_color,
                NULL);

/* Set the values for Light 1 */
pt_f3d.x = -1.5;
pt_f3d.y = -1.0;
pt_f3d.z = 0.4;
xgl_object_set(light[1],
                XGL_LIGHT_TYPE, XGL_LIGHT_DIRECTIONAL,
                XGL_LIGHT_COLOR, &light_color,
                XGL_LIGHT_DIRECTION, &pt_f3d,
                NULL);

/* Turn Light 1 off */
light_switch[1] = FALSE;
xgl_object_set (ctx, XGL_3D_CTX_LIGHT_SWITCHES, light_switch,
                 NULL);
```

15

Light Operators and Attributes

Light Operator

An application can copy a Light object to another Light object. The `xgl_light_copy()` operator makes a copy of a Light object by transferring all state information from a Source Light to a Destination Light. This operator is defined as:

```
void xgl_light_copy (Xgl_light dest_light,Xgl_light src_light)
```

Light Attributes

Most Light attributes in the following list were introduced in the previous section as mathematical symbols used in the lighting equations. More information is available about each of these attributes in the *XGL Reference Manual*.

Table 15-3 Light Object Attributes

Attribute	Description
XGL_LIGHT_TYPE	Defines the type of a Light object. The type can be XGL_LIGHT_AMBIENT, XGL_LIGHT_SPOT, XGL_LIGHT_DIRECTIONAL, or XGL_LIGHT_POSITIONAL.
XGL_LIGHT_POSITION	Defines the position of a light.
XGL_LIGHT_DIRECTION	Defines the direction of a light.
XGL_LIGHT_COLOR	Defines the color of a light.
XGL_LIGHT_SPOT_ANGLE	Defines the illumination angle of influence for a spot light.
XGL_LIGHT_SPOT_EXPONENT	Defines the light source concentration exponent for a spot light.
XGL_LIGHT_ATTENUATION_1 XGL_LIGHT_ATTENUATION_2	Define the light attenuation coefficients of a Light object. These attributes apply to positional lights or spot lights.

3D Context Attributes for Lighting

Several 3D Context surface attributes supply the values for the different variables of the lighting equations. These attributes define the surface appearance of 3D objects. Some were introduced previously as mathematical symbols used in the lighting equations. Information is available on the use of each of these attributes in the *XGL Reference Manual*. Table 15-4 provides an overview of the Context surface rendering attributes.

Table 15-4 Context Light Attributes

Attribute	**Description**
XGL_CTX_FRONT_COLOR XGL_3D_CTX_SURF_BACK_COLOR	Define the color used to fill surfaces.
XGL_CTX_SURF_FRONT_COLOR_SELECTOR XGL_3D_CTX_SURF_BACK_COLOR_SELECTOR	Select the source of a surface's color from the Context or from facet or vertex data provided with the primitive.
XGL_3D_CTX_SURF_FRONT_AMBIENT XGL_3D_CTX_SURF_BACK_AMBIENT	Define the ambient reflection coefficients used for lighting calculations within a 3D Context.
XGL_3D_CTX_SURF_FRONT_DIFFUSE XGL_3D_CTX_SURF_BACK_DIFFUSE	Define the diffuse reflection coefficients used for lighting calculations within a 3D Context.
XGL_3D_CTX_SURF_FRONT_ILLUMINATION XGL_3D_CTX_SURF_BACK_ILLUMINATION	Specify how illumination calculations are performed.
XGL_3D_CTX_SURF_FRONT_LIGHT_ COMPONENT XGL_3D_CTX_SURF_BACK_LIGHT_COMPONENT	Determine which components of the lighting equation are used to compute a surface color.
XGL_3D_CTX_SURF_FRONT_SPECULAR XGL_3D_CTX_SURF_BACK_SPECULAR	Define the specular coefficient used in computing lighting.
XGL_3D_CTX_SURF_FRONT_SPECULAR_COLOR XGL_3D_CTX_SURF_BACK_SPECULAR_COLOR	Define the specular color used in computing lighting.
XGL_3D_CTX_SURF_FRONT_SPECULAR_POWER XGL_3D_CTX_SURF_BACK_SPECULAR_POWER	Defines the specular power used in computing lighting.
XGL_3D_CTX_SURF_FACE_CULL	Controls face culling in a 3D Context.
XGL_3D_CTX_SURF_FACE_DISTINGUISH	Controls whether front- and back-facing surfaces are distinguished from each other.
XGL_3D_CTX_SURF_GEOM_NORMAL	Controls how surface normals are calculated in a Context if the surface normals are not provided as part of the application data.

Table 15-4 Context Light Attributes *(Continued)*

Attribute	Description
XGL_3D_CTX_SURF_NORMAL_FLIP	Specifies whether vertex and facet normals are flipped.
XGL_3D_CTX_SURF_LIGHTING_NORMAL_FLIP	Defines how surface normals are treated for lighting.
XGL_3D_CTX_LIGHT_SWITCHES	Defines the on and off switches that correspond to the 3D Context's array of lights.
XGL_3D_CTX_LIGHTS	Defines the array of lights in a 3D Context.
XGL_3D_CTX_LIGHT_NUM	Sets the number of lights in a 3D Context.

Depth Cueing

As discussed earlier in this chapter depth cueing is the stage of the rendering pipeline that conceptually follows lighting and shading. Depth cueing assists observers with perception of depth by fading the usual colors by an amount that depends on the distance of a primitive from the observer. The amount of fading increases as the depth increases.

Unlike lighting and shading, depth cueing applies to all primitives when it is enabled, not just surface primitives.

Applications

Applications that use depth cueing often combine this effect with motion through animation. Two common applications of depth cueing are molecular modeling and flight simulation. In the molecular example, the background color is often black. Depth perception can be enhanced by fading the more distant parts of the molecule to black, making them appear dimmer. Oscillation of the molecule about a vertical axis enhances depth perception.

Linear Depth Cueing

The simplest method of depth cueing is to blend the usual color of a point on a primitive with a specified depth cue color. This color is usually the same as the color of the background.

Like lighting, depth cueing is simpler in RGB color space than in indexed color space. The colors of primitives and the depth cue color are in the RGB format. Let C be the usual color of a point on a primitive. Let C_{dc} be the depth cue color. Let z be the normalized depth, which equals 0 at the front of the VDC viewport and 1 at the back. The depth cued color $\tilde{C}$ is a weighted average between the usual color and the depth cue color:

$$\tilde{C} = (1 - z) C + zC_{dc}$$

XGL applies this equation in turn to each of the three channels (red, green, and blue).

On indexed Rasters, depth cueing has limitations, just as lighting does. Indexed depth cueing takes place within color ramps. XGL finds the color ramp in which the undepth-cued color index resides and blends this with the color index at the bottom of the color ramp. Let I be the usual color index on a point of a primitive. Let I_0 be the color index at the bottom of the color ramp in which I resides. Let z be the normalized depth, as in the RGB case above. The depth-cued color is the weighted-average:

$$\tilde{I} = (1 - z) I + zI_0$$

In linear depth cueing, the colors of primitives are the same as the “undepth-cued” colors at the front of the VDC viewport. At the back of the VDC viewport, the colors completely change to the specified depth cue color (RGB) or the color at the bottom of the color ramp (indexed). Between these boundaries, all colors modulate to a linear combination of the original color and the depth cue color.

Scaled Depth Cueing

Scaled depth cueing is a superset of linear depth cueing. Scaled depth cueing is linear between two reference planes, which can be placed at arbitrary values of depth in VDC. The scale factors at the front and back planes can be specified as values in the range [0.0, 1.0]. The scale factors outside of the linear region are clamped at the values specified at the reference planes.

The equations for scaled depth cueing are as follows:

$$(C' = S_f C + (1 - S_f)\, C_d) \quad Z \le Z_f$$

$$(C' = S_b C + (1 - S_b)\, C_d) \quad Z \ge Z_b$$

$$(C' = \lambda C + (1 - \lambda)\, C_d) \quad Z_f \le Z \le Z_b$$

$$\lambda = S_b + (S_f - S_b)\left(\frac{Z - Z_b}{Z_f - Z_b}\right)$$

where C' is the depth-cued color, C is the original color, and C_d is the depth-cue color. These equations for illustrative purposes assume Z increases from front to back. Figure 15-1 graphs the scaling factor in scaled depth cueing.

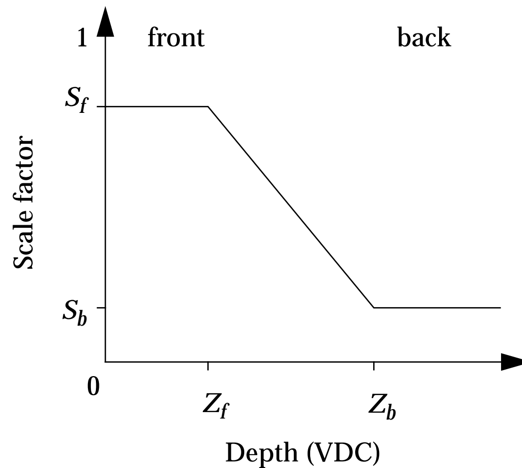


Figure 15-1 Scaling Factor Applied in Scaled Depth Cueing

Depth Cueing Attributes

`XGL_3D_CTX_DEPTH_CUE_MODE`
Selects one of three depth cue modes: `XGL_DEPTH_CUE_LINEAR`, which applies a linear depth cueing function; `XGL_DEPTH_CUE_SCALED`, which applies a linear depth cueing function within the limits of two reference planes; and `XGL_DEPTH_CUE_OFF`. The default value is `XGL_DEPTH_CUE_OFF`.

XGL_3D_CTX_DEPTH_CUE_INTERP
Specifies whether depth cue interpolation is calculated along the depth axis. If the value is set to TRUE, linear interpolation is performed along the primitive for each pixel rendered, using the color defined at each vertex of the primitive by the attribute XGL_3D_CTX_DEPTH_CUE_MODE. If the value is set to FALSE, depth cue color is calculated at each vertex, but no interpolation is done at rendering time. The attribute is assumed to be TRUE if either of the following conditions is true:

- Polylines are being rendered, and XGL_3D_CTX_LINE_COLOR_INTERP is TRUE.
- Surfaces are being rendered, and the attribute controlling illumination on the surface (either XGL_3D_CTX_SURF_BACK_ILLUMINATION or XGL_3D_CTX_SURF_FRONT_ILLUMINATION) is set to XGL_ILLUM_NONE_INTERP_COLOR or XGL_ILLUM_PER_VERTEX.

In other words, if per-pixel interpolated shading has already been requested by the setting of other Context attributes, the XGL_3D_CTX_DEPTH_CUE_INTERP attribute has no effect on rendering. The default value is TRUE.

XGL_3D_CTX_DEPTH_CUE_REF_PLANES
Defines two planes normal to the *z*-axis in VDC that are used when the depth cue mode is set to XGL_DEPTH_CUE_SCALED. In this case, the depth cueing interpolation is performed only on that part of the image that lies between the two depth cue reference planes. The default value is (0.0, 1.0).

XGL_3D_CTX_DEPTH_CUE_SCALE_FACTORS
Defines the portion of a primitive's color that is mixed with the depth cue color when the depth cue mode is set to XGL_DEPTH_CUE_SCALED. Two scale factors are used, one for the part of the primitive in front of the frontmost depth cue reference plane, and one for the part behind the rear-most plane. The default value is (1.0, 0.0).

XGL_3D_CTX_DEPTH_CUE_COLOR
Specifies the color that depth-cued primitives will tend toward for increasing values of Z. The color type of this attribute must match the color type of the Raster attached to the Context. The default value is index 0 or RGB (0, 0, 0).

15

Lighting Examples

The following examples illustrate lighting and shading for an indexed Raster. The main procedure is `light_main.c` in Appendix B. It initializes the data structures that define an icosahedron suitable for the `xgl_multi_simple_polygon()` primitive. It also initializes some XGL resources and the window system interface using the standard example utility file `ex_utils.c`. To compile the light example programs, type `make light`. The complete program includes `ex_utils.c`, `light_main.c`, `light_facet.c`, and `light_vertex.c`; it allows you to view a multi-sided polygon with facet lighting and vertex lighting.

Light Facet Example

This example illustrates illumination on a per-facet basis, also known as flat shading. It sets up two color ramps, initializes the View Transform, defines and turns on three lights, and draws the icosahedron. The output is shown in Plate 5a. Note that the first light that the program defines is an ambient light.

Code Example 15-1 Facet Lighting Example

```
/*
 * light_facet.c
 */
#include <xview/xview.h>
#include <xgl/xgl.h>
#include <math.h>

#include "ex.h"
#include "light.h"

extern Xgl_pt_list   pl[NUM_FACETS];
extern Xgl_color_typeex_color_type;

void
light_facet (Xgl_object    ctx)
{
    Xgl_usgn32           i;
    Xgl_object           ras;
    Xgl_cmap             cmap;
    Xgl_color_list       clist;
    Xgl_color            ctable[CTABLE_NUM];
    Xgl_segment          segment[2];
```

```
Xgl_pt                  pt;
Xgl_bounds_d3d          vdc_window;
Xgl_pt_f3d              pt_f3d;
Xgl_pt_d3d              pt_d3d;
Xgl_object              trans;
Xgl_object              view_trans;
Xgl_color               cyan;
Xgl_light               light[4];
Xgl_boolean             light_switch[4];
Xgl_color               light_color;

if (ex_color_type == XGL_COLOR_INDEX) {
    /* Get the context's raster, and its color map */
    xgl_object_get (ctx, XGL_CTX_DEVICE, &ras);
    xgl_object_get (ras, XGL_RAS_COLOR_MAP, &cmap);

    /* Destroy the old color map, as it will be replaced */
    xgl_object_destroy (cmap);

    /* Set up a color map with two color ramps */

    /* Gray ramp */
    for (i = 0; i < GRAY_LENGTH; i++) {
   ctable[i + GRAY_OFFSET].rgb.r =
   ctable[i + GRAY_OFFSET].rgb.g =
   ctable[i + GRAY_OFFSET].rgb.b =
       (float) i / (float) (GRAY_LENGTH - 1);
    }

    /* Cyan ramp */
    for (i = 0; i < CYAN_LENGTH; i++) {
   ctable[i + CYAN_OFFSET].rgb.r = 0.0;
   ctable[i + CYAN_OFFSET].rgb.g =
   ctable[i + CYAN_OFFSET].rgb.b =
  (float) i / (float) (CYAN_LENGTH - 1);
    }

    /* Color-list data */
    clist.start_index = 0;
    clist.length = CTABLE_NUM;
    clist.colors = ctable;

    /* Color-ramp data */
    segment[GRAY_SEG].offset = GRAY_OFFSET;
    segment[GRAY_SEG].length = GRAY_LENGTH;
    segment[CYAN_SEG].offset = CYAN_OFFSET;
```

```
    segment[CYAN_SEG].length = CYAN_LENGTH;

    /* Create the new color map with the gray and cyan ramps */
   cmap = xgl_object_create (sys_st, XGL_CMAP, NULL,
             XGL_CMAP_COLOR_TABLE_SIZE, CTABLE_NUM,
             XGL_CMAP_COLOR_TABLE, &clist,
             XGL_CMAP_RAMP_NUM, 2,
             XGL_CMAP_RAMP_LIST, segment,
             NULL);

    /* Associate the new color map with the raster */
    xgl_object_set (ras, XGL_RAS_COLOR_MAP, cmap, NULL);
}

/* Clear window since color map size has increased */
xgl_context_new_frame (ctx);

/* Create a transform */
trans = xgl_object_create (sys_st, XGL_TRANS, NULL, NULL);

/* Set up view transform */
pt.pt_type = XGL_PT_F3D;
pt.pt.f3d = &pt_f3d;
pt_f3d.x = 0.0;
pt_f3d.y = -11.0;
pt_f3d.z = -3.0;
xgl_transform_translate (trans, &pt, XGL_TRANS_REPLACE);
xgl_transform_rotate (trans, -PI / 2, XGL_AXIS_X,
        XGL_TRANS_POSTCONCAT);
xgl_transform_rotate (trans, atan (pt_f3d.x / pt_f3d.y) + PI,
        XGL_AXIS_Y, XGL_TRANS_POSTCONCAT);
xgl_transform_rotate (trans,
        atan (-pt_f3d.z /
        sqrt (pt_f3d.x * pt_f3d.x +
        pt_f3d.y * pt_f3d.y)),
        XGL_AXIS_X, XGL_TRANS_POSTCONCAT);

/* Set context's view transform */
xgl_object_get (ctx, XGL_CTX_VIEW_TRANS, &view_trans);
xgl_transform_copy (view_trans, trans);

/* Set up VDC window */
vdc_window.xmin = -1.25;
vdc_window.xmax = 1.25;
vdc_window.ymin = -1.25;
vdc_window.ymax = 1.25;
```

```
vdc_window.zmin = -13.0;
vdc_window.zmax = -9.0;

if (ex_color_type == XGL_COLOR_INDEX) {
    /* Set up material color of polyhedron */
    cyan.index = CYAN_OFFSET + CYAN_LENGTH - 1;
}
else {
    cyan = cyan_color;
}

/* Set context for viewing facet-illuminated polyhedron */
xgl_object_set (ctx,
       XGL_CTX_VDC_ORIENTATION, XGL_Y_UP_Z_TOWARD,
       XGL_CTX_VDC_MAP, XGL_VDC_MAP_ASPECT,
       XGL_CTX_VDC_WINDOW, &vdc_window,
       XGL_3D_CTX_SURF_FACE_CULL, XGL_CULL_BACK,
       XGL_CTX_SURF_FRONT_COLOR, &cyan,
       XGL_3D_CTX_SURF_FRONT_ILLUMINATION,
       XGL_ILLUM_PER_FACET,
       XGL_3D_CTX_SURF_FRONT_AMBIENT, 0.3,
       XGL_3D_CTX_SURF_FRONT_DIFFUSE, 0.7,
       XGL_3D_CTX_SURF_FRONT_SPECULAR, 0.3,
       XGL_3D_CTX_SURF_FRONT_SPECULAR_COLOR, &cyan,
       XGL_3D_CTX_SURF_FRONT_SPECULAR_POWER, 5.0,
       XGL_3D_CTX_SURF_FRONT_LIGHT_COMPONENT,
       XGL_LIGHT_ENABLE_COMP_AMBIENT |
       XGL_LIGHT_ENABLE_COMP_DIFFUSE |
       XGL_LIGHT_ENABLE_COMP_SPECULAR,
       XGL_3D_CTX_LIGHT_NUM, 4,
       NULL);

/* Get lights and switches */
xgl_object_get (ctx, XGL_3D_CTX_LIGHTS, light);
xgl_object_get (ctx, XGL_3D_CTX_LIGHT_SWITCHES, light_switch);

/* Light 0: ambient */
if (ex_color_type == XGL_COLOR_INDEX)
    light_color.gray = 0.95;
else
    light_color.rgb.r = light_color.rgb.g =
          light_color.rgb.b = 0.95;
xgl_object_set (light[0],
       XGL_LIGHT_TYPE, XGL_LIGHT_AMBIENT,
       XGL_LIGHT_COLOR, &light_color,
       NULL);
```

```
    /* Light 1: directional */
    if (ex_color_type == XGL_COLOR_INDEX)
        light_color.gray = 0.85;
    else
        light_color.rgb.r = light_color.rgb.g =
              light_color.rgb.b = 0.85;
    pt_f3d.x = -1.5;
    pt_f3d.y = -1.0;
    pt_f3d.z = 0.4;
    xgl_object_set (light[1],
           XGL_LIGHT_TYPE, XGL_LIGHT_DIRECTIONAL,
           XGL_LIGHT_COLOR, &light_color,
           XGL_LIGHT_DIRECTION, &pt_f3d,
           NULL);

    /* Light 2: positional */
    if (ex_color_type == XGL_COLOR_INDEX)
        light_color.gray = 0.7;
    else
        light_color.rgb.r = light_color.rgb.g =
              light_color.rgb.b = 0.7;
    pt_d3d.x = -5.0;
    pt_d3d.y = 3.0;
    pt_d3d.z = 1.0;
    xgl_object_set (light[2],
           XGL_LIGHT_TYPE, XGL_LIGHT_POSITIONAL,
           XGL_LIGHT_COLOR, &light_color,
           XGL_LIGHT_POSITION, &pt_d3d,
           NULL);

    /* Turn Light 3 off; leave others on */
    light_switch[3] = FALSE;
    xgl_object_set (ctx, XGL_3D_CTX_LIGHT_SWITCHES, light_switch,
           NULL);

    /* Draw facet-illuminated polyhedron */
    xgl_multi_simple_polygon (ctx,
                 XGL_FACET_FLAG_SIDES_ARE_3 |
                 XGL_FACET_FLAG_SHAPE_CONVEX,
                 NULL, NULL, NUM_FACETS, pl);

    /* Clean up */
    xgl_object_destroy (trans);
}
```

Light Vertex Example

This example illustrates lighting on a per-vertex basis. It uses the color ramps, View Transform, and Lights defined in the previous example. In addition, it defines one additional Light, turns it on, and redraws the icosahedron with the other color ramp and vertex illumination. The output is shown in Plate 5b.

Code Example 15-2 Vertex Lighting Example

```
/*
 * light_vertex.c
 */
#include <xview/xview.h>
#include <xgl/xgl.h>

#include "ex.h"
#include "light.h"

extern Xgl_pt_list  pl[NUM_FACETS];
extern Xgl_color_typeex_color_type;

void
light_vertex (Xgl_object   ctx)
{
    Xgl_color            gray;
    Xgl_object           light[4];
    Xgl_boolean          light_switch[4];
    Xgl_color            light_color;
    Xgl_pt_f3d           pt_f3d;
    Xgl_pt_d3d           pt_d3d;

    /* Get the list of lights and switches */
    xgl_object_get (ctx, XGL_3D_CTX_LIGHTS, light);
    xgl_object_get (ctx, XGL_3D_CTX_LIGHT_SWITCHES, light_switch);

    /* Light 3: spot */
    if (ex_color_type == XGL_COLOR_INDEX)
       light_color.gray = 0.2;
    else
       light_color.rgb.r =
       light_color.rgb.g =
       light_color.rgb.b = 0.2;
    pt_d3d.x = 1.0;
    pt_d3d.y = 15.0;
```

```
    pt_d3d.z = 2.0;
    xgl_object_set (light[3],
           XGL_LIGHT_TYPE, XGL_LIGHT_SPOT,
           XGL_LIGHT_COLOR, &light_color,
           XGL_LIGHT_POSITION, &pt_d3d,
           NULL);
    pt_f3d.x = -1.0;
    pt_f3d.y = -15.0;
    pt_f3d.z = -2.0;
    xgl_object_set (light[3],
           XGL_LIGHT_DIRECTION, &pt_f3d,
           XGL_LIGHT_SPOT_ANGLE, PI / 20,
           XGL_LIGHT_SPOT_EXPONENT, 4.0,
           NULL);

    /* Turn Light 3 on */
    light_switch[3] = TRUE;
    xgl_object_set (ctx,
   XGL_3D_CTX_LIGHT_SWITCHES, light_switch, NULL);

    /* Set up context for Gouraud-shaded polygons */
    if (ex_color_type == XGL_COLOR_INDEX)
        gray.index = GRAY_OFFSET + GRAY_LENGTH - 1;
    else
        gray.rgb.r = gray.rgb.g = gray.rgb.b = 1.0;
    xgl_object_set (ctx,
          XGL_3D_CTX_SURF_FRONT_ILLUMINATION, XGL_ILLUM_PER_VERTEX,
           XGL_CTX_SURF_FRONT_COLOR, &gray,
           XGL_3D_CTX_SURF_FRONT_SPECULAR_COLOR, &gray,
           NULL);

/* Draw polyhedron with vertex-illuminated,Gouraud-shaded facets */
    xgl_multi_simple_polygon (ctx,
                 XGL_FACET_FLAG_SIDES_ARE_3 |
                 XGL_FACET_FLAG_SHAPE_CONVEX,
                 NULL, NULL, NUM_FACETS, pl);
}
```

Linear Depth Cueing Example

This example illustrates linear depth cueing. This type of depth cueing changes the colors of primitives from the front to the back of the VDC view space linearly. To compile this program, type `make dcue` in the example programs directory. The complete program includes the `ex_utils.c` utility file listed in Appendix B, `dcue_main.c` listed in Appendix B, and the linear and scaled depth cueing example programs listed in this chapter.

Code Example 15-3 Linear Depth Cueing Example

```
/*
 * dcue_linear.c
 */
#include <xview/xview.h>
#include <xgl/xgl.h>

#include "ex.h"
#include "dcue.h"

void
dcue_linear (Xgl_object   ctx)
{
    Xgl_object           ras;   /* raster associated with ctx */
    Xgl_object           cmap;  /* color map associated ras */
    Xgl_color            ln_color;      /* vector color */
    Xgl_object           view_trans;

    /*
     * get raster object from context
     */
    xgl_object_get (ctx, XGL_CTX_DEVICE, &ras);

    if (ex_color_type == XGL_COLOR_INDEX) {

   /*
   * get color map object from raster
   */
   xgl_object_get (ras, XGL_RAS_COLOR_MAP, &cmap);

   /*
    * if color map associated with raster isn't rampcmap then
    * put it into the raster
    */
```

```
if (cmap != rampcmap) {

   if (!rampcmap) {
   Xgl_sgn32          i;
   Xgl_color_list     clist;      /* color table list */
   Xgl_color          colors[RAMP_SIZE];/* list of colors */
   Xgl_segment        segment;          /* color ramp segment */

   /*
    * create color list
    */
   clist.start_index = 0;
   clist.length      = RAMP_SIZE;
   clist.colors      = colors;

   /*
    * gray ramp for depth cued vectors
    */
   for (i = 0; i < RAMP_SIZE; i++) {
       colors[i].rgb.r =
       colors[i].rgb.g =
       colors[i].rgb.b = ((float) i) / ((float)(RAMP_SIZE - 1));
   }

   /*
    * set up segment data
    */
   segment.offset = 0;
   segment.length = RAMP_SIZE;

   /*
    * create and set attributes for ramp color map
    */
   rampcmap = xgl_object_create(sys_st, XGL_CMAP, NULL,
                XGL_CMAP_COLOR_TABLE_SIZE, RAMP_SIZE,
                XGL_CMAP_COLOR_TABLE, &clist,
                XGL_CMAP_RAMP_NUM, 1,
                XGL_CMAP_RAMP_LIST, &segment,
                NULL);
   }
    /*
     * set color map in raster
     */
    xgl_object_set (ras, XGL_RAS_COLOR_MAP, rampcmap, NULL);
   }
} else {/* RGB raster */
```

```
        printf("no ramps on RGB rasters\n");
    }

    /*
     * modify the viewing transform so we can see the cube
     */
    xgl_object_get (ctx, XGL_CTX_VIEW_TRANS, &view_trans);
    xgl_transform_rotate (view_trans, 0.3,
                     XGL_AXIS_Z, XGL_TRANS_POSTCONCAT);
    xgl_transform_rotate (view_trans, 0.3,
                     XGL_AXIS_Y, XGL_TRANS_POSTCONCAT);
    xgl_transform_rotate (view_trans, 0.3,
                     XGL_AXIS_X, XGL_TRANS_POSTCONCAT);

    /*
     * render a depth cued multipolyline
     */
   xgl_object_set (ctx, XGL_CTX_VDC_MAP, XGL_VDC_MAP_DEVICE, NULL);
    xgl_context_new_frame (ctx);
   xgl_object_set (ctx, XGL_CTX_VDC_MAP, XGL_VDC_MAP_ASPECT, NULL);

    xgl_object_set (ctx, XGL_3D_CTX_DEPTH_CUE_MODE,
                   XGL_DEPTH_CUE_LINEAR, NULL);

    if (ex_color_type == XGL_COLOR_INDEX)
       /* set color to highest value in ramp */
       ln_color.index = RAMP_SIZE - 1;
    else
       ln_color = white_color;

    xgl_object_set (ctx, XGL_CTX_LINE_COLOR, &ln_color, NULL);

    xgl_multipolyline (ctx, NULL, 1, &pl_3d);

    /* display color table */
    color_show (ras, -1);

    /* restore viewing transformation */
    xgl_transform_identity (view_trans);

    /* turn off depth cueing */
    xgl_object_set (ctx, XGL_3D_CTX_DEPTH_CUE_MODE,
                   XGL_DEPTH_CUE_OFF, NULL);
}
```

Scaled Depth Cueing Example

This example illustrates scaled depth cueing. This type of depth cueing is controlled by the application and occurs only between the two reference planes that are specified. It attenuates the colors of the primitives only to the amount specified.

Code Example 15-4 Scaled Depth Cueing Example

```
/*
 * dcue_scaled.c
 */
#include <xview/xview.h>
#include <xgl/xgl.h>

#include "ex.h"
#include "dcue.h"

void
dcue_scaled (Xgl_object  ctx)
{
    Xgl_object     ras;  /* raster associated with ctx */
    Xgl_object     cmap; /* color map associated with ras */
    Xgl_color      ln_color;/* vector color */
    Xgl_object     view_trans;

    Xgl_bounds_d3d vdc_window;     /* VDC window coords */
    double         saved_zmn, saved_zmx;    /* saved VDC coords */
    double         ref_planes[2];  /* scaled dcue reference planes */
    float          scale_factors[2]; /* scaled dcue scaling factors */

    /*
     * get raster object from context
     */
    xgl_object_get (ctx, XGL_CTX_DEVICE, &ras);

    if (ex_color_type == XGL_COLOR_INDEX) {

    /*
     * get color map object from raster
     */
    xgl_object_get (ras, XGL_RAS_COLOR_MAP, &cmap);

    /*
```

```
 * if color map associated with raster isn't rampcmap then
 * put it into the raster
 */
if (cmap != rampcmap) {

   if (!rampcmap) {
   Xgl_sgn32          i;
   Xgl_color_list     clist;      /* color table list */
   Xgl_color          colors[RAMP_SIZE];/* list of colors */
   Xgl_segment        segment;    /* color ramp segment */

   /*
    * create color list
    */
   clist.start_index = 0;
   clist.length      = RAMP_SIZE;
   clist.colors      = colors;

   /*
    * gray ramp for depth cued vectors
    */
   for (i = 0; i < RAMP_SIZE; i++) {
       colors[i].rgb.r =
       colors[i].rgb.g =
       colors[i].rgb.b = ((float) i) / ((float)(RAMP_SIZE - 1));
   }

   /*
    * setup segment data
    */
   segment.offset = 0;
   segment.length = RAMP_SIZE;

   /*
    * create and set attributes for ramp color map
    */
   rampcmap = xgl_object_create(sys_st, XGL_CMAP, NULL,
               XGL_CMAP_COLOR_TABLE_SIZE, RAMP_SIZE,
               XGL_CMAP_COLOR_TABLE, &clist,
               XGL_CMAP_RAMP_NUM, 1,
               XGL_CMAP_RAMP_LIST, &segment,
               NULL);
   }

   /*
    * set color map in raster
```

```
        */
      xgl_object_set (ras, XGL_RAS_COLOR_MAP, rampcmap, NULL);
    }
  } else {/* RGB raster */
        printf("no ramps on RGB rasters\n");
  }

  /*
   * modify the viewing transform so we can see the cube
   */
  xgl_object_get (ctx, XGL_CTX_VIEW_TRANS, &view_trans);

  xgl_transform_rotate (view_trans, 0.3,
          XGL_AXIS_Z, XGL_TRANS_POSTCONCAT);

  xgl_transform_rotate (view_trans, 0.3,
          XGL_AXIS_Y, XGL_TRANS_POSTCONCAT);

  xgl_transform_rotate (view_trans, 0.3,
          XGL_AXIS_X, XGL_TRANS_POSTCONCAT);

  /*
   * render a depth cued multipolyline
   */
 xgl_object_set (ctx, XGL_CTX_VDC_MAP, XGL_VDC_MAP_DEVICE, NULL);
  xgl_context_new_frame (ctx);
 xgl_object_set (ctx, XGL_CTX_VDC_MAP, XGL_VDC_MAP_ASPECT, NULL);

  /* get VDC window and set VDC z values */
  xgl_object_get (ctx, XGL_CTX_VDC_WINDOW, &vdc_window);
  GET_VDC_Z(saved_zmn, saved_zmx)
  SET_VDC_Z(-1.0, 1.0)

 /* set reference planes and scale factors for scaled depth cueing
*/
  ref_planes[0] = 0.1;
  ref_planes[1] = 0.3;
  scale_factors[0] = 0.3;
  scale_factors[1] = 1.0;
  xgl_object_set (ctx,
           XGL_3D_CTX_DEPTH_CUE_SCALE_FACTORS, scale_factors,
             XGL_3D_CTX_DEPTH_CUE_REF_PLANES, ref_planes,
             XGL_CTX_VDC_WINDOW, &vdc_window,
             NULL);
```

```
    /* turn on scaled depth cueing */
    xgl_object_set (ctx, XGL_3D_CTX_DEPTH_CUE_MODE,
                   XGL_DEPTH_CUE_SCALED, NULL);

    if (ex_color_type == XGL_COLOR_INDEX)
       /* set color to highest value in ramp */
       ln_color.index = RAMP_SIZE - 1;
    else
       ln_color = white_color;
    xgl_object_set (ctx, XGL_CTX_LINE_COLOR, &ln_color, NULL);

    /* draw cube */
    xgl_multipolyline (ctx, NULL, 1, &pl_3d);

    /* display color table */
    color_show (ras, -1);

    /* restore VDC window */
    SET_VDC_Z(saved_zmn, saved_zmx)
    xgl_object_set (ctx, XGL_CTX_VDC_WINDOW, &vdc_window, NULL);

    /* restore viewing transformation */
    xgl_transform_identity (view_trans);

    /* turn off depth cueing */
    xgl_object_set (ctx, XGL_3D_CTX_DEPTH_CUE_MODE,
                   XGL_DEPTH_CUE_OFF, NULL);
}
```

Caching Geometry 16

This chapter discusses the the display list capability provided by the XGL Pcache object and the geometry cache function provided by the XGL Gcache object. It includes information on the following topics:

- How to create and display geometry using the Pcache and Gcache objects
- Performance considerations for Pcache and Gcache use
- Gcache operators and attributes

Introduction to Geometry Caching in XGL

The XGL library provides applications with two basic buffering mechanisms to cache the geometric representation of data. One mechanism, the Pcache object, provides non-editable, non-hierarchical display list functionality for XGL. The Pcache object stores a sequence of primitives and relevant attributes for rendering at one time. Using the Pcache object, application programmers can write XGL code and tune it for the high performance that display lists can provide.

The second mechanism, the Gcache object, provides a facility to decompose complex primitives and accelerate their rendering through the use of simpler primitives better suited for a particular graphics device. The Gcache object reduces the complexity of a primitive by allowing XGL to process the primitive into many simple primitives, storing the relevant attribute values. For example, a Gcache could contain the polylines that comprise a string of stroke text, or

the triangles that make up a polygon. Both Pcache and Gcache objects provide a mechanism to store graphics primitives that are rendered frequently, thus yielding higher performance.

The XGL library allows applications to choose between immediate mode, Pcache programming, or Gcache programming. It permits the application to manage its graphics geometry and to decide what information is kept in the Pcache or Gcache object without imposing constraints. Pcache or Gcache programming is slightly different than immediate mode XGL programming. Primitives are rendered into these objects rather than directly to the display device. When using a Pcache or Gcache object, the application follows these general steps:

1. Create a Context object.
2. Create a Pcache or Gcache object. If the object is a Pcache object, associate it with the Context object.
3. Render the geometry into the Pcache or Gcache.
4. Display the Pcache or Gcache.

The following sections discuss the characteristics and use of these objects in more detail.

Pcache Object

A Primitive Cache (Pcache) is an XGL object that stores a sequence of zero or more XGL primitives and/or attribute settings. Pcache contents can be stored in a device dependent format in order to maximize display speed.

The use of Pcache objects optimizes performance for most 3D graphics applications running on graphics adaptors with display list capabilities. A Pcache can be rendered with greater speed, in general, than its individual components can. Several characteristics of the Pcache contribute to its optimizing effect:

- Once a Pcache is created, it cannot be modified. This reduces overhead that would be associated with searching an editable cache, and with memory management.

- Commands are stored in a format that is optimal for the graphics hardware. Optimizing commands for the particular hardware increases performance in either a networked or a local environment.
- Values of coordinates and variables are copied into the Pcache when it is compiled. This saves on processing overhead. The more times a Pcache is displayed, the greater the performance payoff.

Using Pcache Objects With Immediate Mode Applications

Many application developers may be using display list programming models even though their code is written in immediate mode. For example, an immediate mode application may use a preformatted display list that contains all the data associated with the current model. Immediate mode applications like this, and applications rendering geometry that does not change from frame to frame, could consider using Pcache objects to gain in rendring performance.

Applications that can benefit from Pcache programming range from visual simulation to mechanical computer-aided engineering to seismic interpretation. A good example of an immediate mode application is a visual simulation application, where an entire scene is rendered per frame. In this type of application, what changes from scene to scene is the relative position of each object and the point of view. Single objects in the scene could easily be encapsulated into a Pcache. For example, in a driving simulation application using an outdoor scene, mountains could be put into one Pcache, a group of buildings in another, and each moving vehicle in its own Pcache. This could dramatically speed up the process of getting all the data transformed for use in the graphics system. Each Pcache could have many computationally intensive operations done before display time, thereby achieving a significant performance improvement.

Performance in these applications can be maximized by knowing how and when to use a Pcache. As the Pcache can be used in conjunction with immediate mode commands, there is no associated penalty in moving to the Pcache model of programming.

Pcache Performance Considerations

There are some important performance trade-offs that should be considered when using Pcache functionality. In immediate mode, geometries are rendered as they are defined. When using Pcache, geometries are first rendered into the Pcache object and later displayed using the `xgl_pcache_display()` function. There is a one-time build cost associated with the Pcache. The greater the number of times a Pcache is displayed, the lower the impact of this overhead on application performance. In this case, the optimizations described above give better performance than that available through immediate mode.

Creating and Rendering to a Pcache

A Pcache object is created with the `xgl_object_create()` operator, using `XGL_PCACHE` as the *type* parameter value, as in the following example:

```
pcache = xgl_object_create(sys_st, XGL_PCACHE, NULL, NULL);
```

To render into the Pcache, the Pcache object must be associated with a 2D or 3D Context object using the `XGL_PCACHE_CONTEXT` attribute. When `XGL_PCACHE_CONTEXT` is set, the Pcache is emptied, even if the setting is redundant. The default value of `XGL_PCACHE_CONTEXT` is `NULL`.

```
xgl_object_set(pcache, XGL_PCACHE_CONTEXT, ctx, NULL);
```

When the Pcache and the Context are associated, the geometry is rendered using the Pcache object in place of the Context object. After the Pcache is created and bound to a Context and its device, the XGL primitives listed below may render to the Pcache:

- `xgl_multimarker()`
- `xgl_multipolyline()`
- `xgl_stroke_text()`
- `xgl_multi_simple_polygon()`
- `xgl_triangle_list()`
- `xgl_quadrilateral_mesh()`

The attributes listed in Table 16-1 can be rendered by the `xgl_object_set()` command to a Pcache object. The first group, Context line attributes, set line style and line color. Context marker attributes control the appearance of markers, and the Context surface attributes define many aspects of surface rendering. Picking may be done on the contents of the Pcache.

Table 16-1 Context Attributes for the Pcache Object

Attribute type	Attributes
Context line attributes	XGL_CTX_LINE_COLOR XGL_CTX_LINE_STYLE XGL_CTX_LINE_WIDTH_SCALE_FACTOR XGL_CTX_LINE_COLOR_SELECTOR
Context marker attribute	XGL_CTX_MARKER_COLOR_SELECTOR
Context surface attributes	XGL_3D_CTX_SURF_FACE_CULL XGL_CTX_SURF_FRONT_COLOR XGL_CTX_SURF_FRONT_COLOR_SELECTOR XGL_3D_CTX_SURF_FRONT_ILLUMINATION XGL_3D_CTX_SURF_BACK_COLOR XGL_3D_CTX_SURF_BACK_COLOR_SELECTOR XGL_3D_CTX_SURF_BACK_ILLUMINATION XGL_3D_CTX_SURF_GEOM_NORMAL
Picking attributes	XGL_CTX_PICK_ID_1 XGL_CTX_PICK_ID_2

When these attributes are applied to a Pcache, they are recorded in a device-specific format. At a later time, the `xgl_pcache_display()` operator can display its contents. Once `xgl_pcache_display()` is called, the Pcache is closed, and the attributes may no longer be changed.

An application can empty (and re-open) a Pcache by applying the operator `xgl_context_new_frame()`. A Pcache is also emptied whenever it is associated with another Context or its Context is associated with another device. It is an error to use the Pcache (except to destroy or re-associate it) if either its Context or Device has been destroyed.

Displaying a Pcache

The application displays the Pcache using the `xgl_pcache_display()` operator. This function performs four tasks:

1. It closes the Pcache.
2. It compares the Pcache and Context states.
3. If Context state and Pcache state match, it displays the Pcache.
4. It returns the result of the comparison.

```
Xgl_cache_display
xgl_pcache_display(
    Xgl_pcache      pcache,
    Xgl_boolean     test,
    Xgl_boolean     display,
    Xgl_boolean     restore);
```

When a Pcache is built, XGL may make assumptions about the state of the Context to which it is attached. At display time, the application may test these assumptions to verify that they are still valid. If so, the Pcache is rendered. If not, the application is informed that the Pcache needs to be rebuilt. The parameter `test` controls the validation of the Pcache and Context states. If `test` is `FALSE`, the Context state is not validated and the return value is `XGL_CACHE_NOT_CHECKED`. If `test` is `TRUE`, the Context state is compared with that of the Pcache. If the state is compatible, the return value is `XGL_CACHE_DISPLAY_OK`; otherwise, it is `XGL_CACHE_ATTR_STATE_DIFFERENT`.

The `display` parameter controls the display of the Pcache. If `display` is `FALSE`, the Pcache is not rendered. If `display` is `TRUE`, the Pcache is rendered only if test is `FALSE` or the Context state is compatible.

The `restore` parameter controls whether or not Context attributes can be altered by displaying the Pcache. If `restore` is `TRUE`, displaying a Pcache will not change any of the Context attributes. If `restore` is `FALSE` and the Pcache contains attributes, the attributes of the Context are changed to reflect those stored in the Pcache.

The application can remove the Pcache with the the `xgl_object_destroy()` function. The code fragment below shows the creation and use of a Pcache object.

```
(...parameter definition and other code omitted...)

/* Create the context object */
ctx = xgl_object_create(sys_state,
          XGL_3D_CTX, (Xgl_obj_desc *) 0,
          XGL_CTX_DEVICE, ras,
          NULL);

/* Create the Pcache object */
pcache = xgl_object_create(sys_state, XGL_PCACHE, NULL,
          NULL);

/* Associate the Pcache with the ctx */
xgl_object_set(pcache, XGL_PCACHE_CONTEXT, ctx, NULL);

xgl_context_new_frame(ctx);

create_tristrip(&pl, &fl);

/* Render the triangle strip to the Pcache */
xgl_triangle_list(pcache, &fl, &pl,
          XGL_TLIST_FLAG_TRI_STRIP);

/* Now render everything in the Pcache to the ctx */
xgl_pcache_display(pcache, FALSE, TRUE, FALSE);

/* Destroy the Pcache */
xgl_object_destroy(pcache);
```

16

Gcache Object

A Gcache object can be used to to break down complex objects into simpler forms, resulting in increased rendering performance (if the underlying accelerator is optimized for the resulting decomposed primitive). A Gcache also enables an application to instruct XGL to model clip or decompose a primitive once, and then render the result many times.

Once a Gcache object is created, an application can use the Gcache operators to store a primitive into the Gcache. For example, `xgl_gcache_polygon()`, which is used to store a polygon in a Gcache, is the Gcache counterpart of `xgl_polygon()`. The primitive can be displayed repeatedly, usually with better performance than its non-Gcache counterpart, by calling `xgl_context_display_gcache()`.

Whenever an object is stored in a Gcache, certain Context attributes (which depend on the primitive being stored) will be saved in the Gcache. If a primitive is stored in a Gcache, and any relevant context attribute changes, the properties of the cached primitive will be outdated. When this happens, the application can decide if the primitive should still be displayed. Many context attributes can be changed after a primitive has been stored in a Gcache without outdating the primitive. For example, view clipping attributes can be changed, and the primitive will still be rendered correctly.

A Gcache copies the original geometry into internal storage in Model Coordinates. For some reductions (polygon decomposition, for example), it is possible for the Gcache to reference the original geometry. This is useful for saving memory.

When To Use a Gcache

Many graphics hardware accelerators are efficient with simple geometries. For example, an accelerator may draw triangles very fast. If an application wants to render the same complex polygon several times, it is worthwhile to use a Gcache to tessellate the polygon into triangles and send the triangles to the accelerator for each rendering. If a Gcache was not used, then the polygon tessellation would have to be done each time the primitive was rendered. Additionally, normals will be calculated (assuming they were not provided by the application) and stored in the facet structure. This saves the Gcache operator from having to calculate normals every time the primitive is rendered.

For stroke text, a Gcache uses the character attributes (character spacing, height, and so on) to convert the text to a set of polylines for faster rendering.

For NURBS curves and surfaces, much of the processing time is absorbed during Gcache creation time. This substantially speeds up display of NURBS curves and surfaces.

Creating a Gcache

A Gcache object is created with the `xgl_object_create()` operator, using `XGL_GCACHE` as the *type* parameter value, as in the following example:

```
gcache = xgl_object_create(sys_st, XGL_GCACHE, NULL, NULL);
```

Rendering to a Gcache

The Gcache object has its own set of operators. These operators use the same data structures as the standard XGL drawing primitives. To render into the Gcache, the application uses a Gcache operator and specifies the Context. The Gcache primitives are:

`xgl_gcache_multi_simple_polygon()`
: This operator stores multisimple polygons in a Gcache. If model clipping is turned on (that is, if `XGL_3D_CTX_MODEL_CLIP_PLANE_NUM` is greater than 0), then the polygons are stored after being model clipped. The polygons are checked to see if they are convex. If a non-convex polygon is found and `XGL_GCACHE_DO_POLYGON_DECOMP` is `TRUE`, XGL tessellates the polygon into triangles. If facet normals are not provided by the application, they are calculated and stored in the Gcache. The tessellation edges can be seen if `XGL_CTX_SURF_EDGE_FLAG` is `TRUE` and `XGL_GCACHE_SHOW_DECOMP_EDGES` is `TRUE`.

`xgl_gcache_multi_elliptical_arc()`
: This operator caches the polyline representation of 3D elliptical arcs in the specified Gcache.

`xgl_gcache_multimarker()`
: This operator stores markers in the specified Gcache.

`xgl_gcache_multipolyline()`
This operator stores 3D polylines and their bounding boxes in the specified Gcache.

`xgl_gcache_nurbs_curve()`
Depending on the value of the Gcache attribute `XGL_GCACHE_NURBS_CURVE_MODE`, the `xgl_gcache_nurbs_curve()` operator stores a NURBS curve in a tessellation-independent form, stores the curve in a tesselated polyline representation, or stores both forms of the curve.

If the value of `XGL_GCACHE_NURBS_CURVE_MODE` is `XGL_GCACHE_NURBS_DYNAMIC`, a tessellation-independent form of the curve is created from the NURBS input parameters, and this form is dynamically tessellated during display. If the value of `XGL_GCACHE_NURBS_CURVE_MODE` is `XGL_GCACHE_NURBS_STATIC`, the curve is tessellated at Gcache creation time, and the polyline representation is stored. If the value of `XGL_GCACHE_NURBS_CURVE_MODE` is `XGL_GCACHE_NURBS_COMBINED`, both the tessellation-independent form of the curve and the static polyline form of the NURBS curve are stored. Portions of the curve are tessellated only if it is necessary according to the current Context; otherwise, the cached tessellation is reused. The application should choose the appropriate representation mode depending on its needs. Refer to the section "Gcache Attributes" on page 442 for a detailed discussion of the `XGL_GCACHE_NURBS_CURVE_MODE` attribute and the three representation modes.

`xgl_gcache_nurbs_surface()`
Depending on the value of the Gcache attribute `XGL_GCACHE_NURBS_SURF_MODE`, the `xgl_gcache_nurbs_surface()` operator stores a NURBS surface in a tessellation-independent form, stores the surface in a tesselated polyline representation, or stores both forms of the surface. The values of the `XGL_GCACHE_NURBS_SURF_MODE` attribute are similar to those of the NURBS curve `XGL_GCACHE_NURBS_CURVE_MODE` attribute.

If the value of `XGL_GCACHE_NURBS_SURF_MODE` is `XGL_GCACHE_NURBS_DYNAMIC`, a tessellation-independent form of the surface is cached, and this form is dynamically tessellated during display. If the value is `XGL_GCACHE_NURBS_STATIC`, the surface is tessellated, and the tessellation is cached. Tessellation consists of a list of triangle lists and/or quad meshes. If isoparametric lines are drawn, they are cached as a list of

polylines. If the value is XGL_GCACHE_NURBS_COMBINED, both forms are cached. Tessellation is recreated where necessary. As in the case of NURBS curves, the application should carefully choose the appropriate representation mode depending on its needs. Refer to the section "Gcache Attributes" on page 442 for a more detailed discussion of the three representation modes.

xgl_gcache_polygon()
: This operator stores 3D polygons in the specified Gcache. If XGL_GCACHE_DO_POLYGON_DECOMP is TRUE, the polygon is broken down into a list of triangles and stored in the Gcache. Additionally, if a facet normal was not provided, the operator will calculate the normal to store in the Gcache.

  Hollow polygons or polygons with edges turned on are drawn using the original edges. The flag XGL_GCACHE_SHOW_DECOMP_EDGES has to be set to TRUE to see the tessellation.

xgl_gcache_stroke_text()
: This operator stores the polyline representation of a text string in the specified Gcache. The Gcache can contain either 2D or 3D data, and this operator accepts both 2D and 3D Contexts. The attribute XGL_GCACHE_USE_APPL_GEOM is ignored for stroke text. Note that the dimension of the Context used to write text to the Gcache must match the dimension of the Context used to display it.

xgl_gcache_triangle_list()
: This operator stores a 3D triangle list in the specified Gcache. If facet normals are not provided, the operator calculates the normals for each facet to store in the Gcache.

xgl_gcache_triangle_strip()
: This operator stores a 3D triangle strip in the specified Gcache. If facet normals are not provided, the operator calculates the normals for each facet to store in the Gcache.

The Context attributes saved in the Gcache for various primitives are listed in the *XGL Reference Manual* under the man page of each primitive.

Gcache Attributes

The Gcache object includes the following attributes:

XGL_GCACHE_BYPASS_MODEL_CLIP
Controls whether model clipping is performed when a Gcache is displayed. If this attribute is FALSE (default mode), model clipping (if enabled in the context) is applied to the cached geometry.

XGL_GCACHE_DISPLAY_PRIM_TYPE
Returns the type of primitive used when the Gcache is displayed. If XGL_GCACHE_DISPLAY_PRIM_TYPE is not the original primitive type, some sort of optimization or simplification was performed on the data by XGL. For example, a complex polygon may have been tessellated into triangles, in which case xgl_multi_simple_polygon() is used for rendering.

XGL_GCACHE_ORIG_PRIM_TYPE
Returns the type of the original primitive stored in the Gcache.

XGL_GCACHE_DO_POLYGON_DECOMP
Controls whether polygon decomposition is performed when xgl_gcache_polygon() is called. If XGL_GCACHE_DO_POLYGON_DECOMP is set to TRUE, any subsequent polygons that are cached will be stored as a decomposed list of triangles. Note that when non-planar polygons are decomposed, the results are undefined.

XGL_GCACHE_SHOW_DECOMP_EDGES
Controls whether decomposition edges are shown when xgl_gcache_polygon() or xgl_gcache_multi_simple_polygon() is displayed. If XGL_GCACHE_SHOW_DECOMP_EDGES and XGL_CTX_SURF_EDGE_FLAG are TRUE, and the Gcache polygon or multisimple polygon was decomposed, the edges outlining the decomposition are shown.

XGL_GCACHE_POLYGON_TYPE
Controls which decomposition algorithm is used when xgl_gcache_polygon() is called.

XGL_GCACHE_PT_LIST_LIST
Returns the list of point lists used when the Gcache is displayed.

XGL_GCACHE_FACET_LIST_LIST
Returns the list of facets used when the Gcache is displayed.

XGL_GCACHE_IS_EMPTY
When set to TRUE (default mode), this attribute empties the Gcache. Nothing is drawn when an empty Gcache is displayed. Gcache resources are not freed until the Gcache object is destroyed.

XGL_GCACHE_USE_APPL_GEOM
When set to TRUE, this attribute causes the Gcache to reference the application's geometry (parameters to the Gcache operators, such as the list of points, bounding box, and facet information) in the Gcache. When set to FALSE (default value), the Gcache copies the applications geometry into the Gcache.

XGL_GCACHE_NURBS_CURVE_MODE
Controls the Gcache mode when the xgl_gcache_nurbs_curve() or xgl_gcache_multi_elliptical_arc() operators are called. This attribute can have one of three values. If the value of XGL_GCACHE_NURBS_CURVE_MODE is XGL_GCACHE_NURBS_DYNAMIC, a tessellation-independent Gcache is created from the NURBS input parameters. At each frame, the curve is tessellated according to the current Context. To obtain dynamic tessellation, the Gcache should be displayed using one of the NURBS curve dynamic approximation types. Memory consumption is minimal in this mode because the tessellation is not stored. In addition, the quality of the displayed geometry is optimal because tessellation step sizes are dynamically computed per frame.

If the value of XGL_GCACHE_NURBS_CURVE_MODE is XGL_GCACHE_NURBS_STATIC, the curve is tessellated at Gcache creation time, and the polyline representation is stored. With this mode, display on some devices may be fast, but memory consumption may be high and the quality of the rendered primitive may be affected by extreme view changes since the stored tessellation cannot be modified. In addition, the Gcache may be invalidated if any NURBS-specific attributes (such as the approximation value) changes.

If the value of XGL_GCACHE_NURBS_CURVE_MODE is XGL_GCACHE_NURBS_COMBINED, both the tessellation-independent form of the curve and the static polyline form of the NURBS curve are stored. Portions of the curve are tessellated only if this is necessary according to the

current Context; otherwise, the cached tessellation is reused. This mode offers the best aspects of the static and dynamic curve representation modes; however, memory consumption may be high.

The application should carefully consider the following when choosing a representation mode:

- Memory availability: Because the static and combined Gcache modes store tessellation, they consume more memory than the dynamic mode.
- Quality of rendered geometry: The dynamic Gcache mode will consistently provide the optimal quality based on the current approximation type, whereas with the other modes, too much or too little tessellation may be displayed.
- Speed: If the hardware device has a fast CPU, or if the device can accelerate NURBS tessellation, dynamic tessellation will be comparable in speed to displaying pre-stored tessellation. Otherwise, in some cases, the application may want to use static or combined Gcache attributes to obtain better performance.
- Complexity of NURBS data and viewing requirements: If the NURBS curve consists of a lot of Bezier points and the application is planning to allow the user to zoom in and out to look closely at the geometry, use dynamic mode. If the NURBS geometry is complex but only a fixed view of it is needed, use static mode. Static mode may store a simpler form of the geometry, thus saving memory and rendering time.

The values of the `XGL_GCACHE_NURBS_CURVE_MODE` are illustrated in Figure 16-1 on page 443.

`XGL_GCACHE_NURBS_SURF_MODE`
Controls the Gcache mode when the `xgl_gcache_nurbs_surface()` operator is called. The values for this attribute are the same as for the `XGL_GCACHE_NURBS_CURVE_MODE` attribute; refer to the preceding section for a description of the three Gcache NURBS surface representation modes and for information on choosing the appropriate mode for an application.

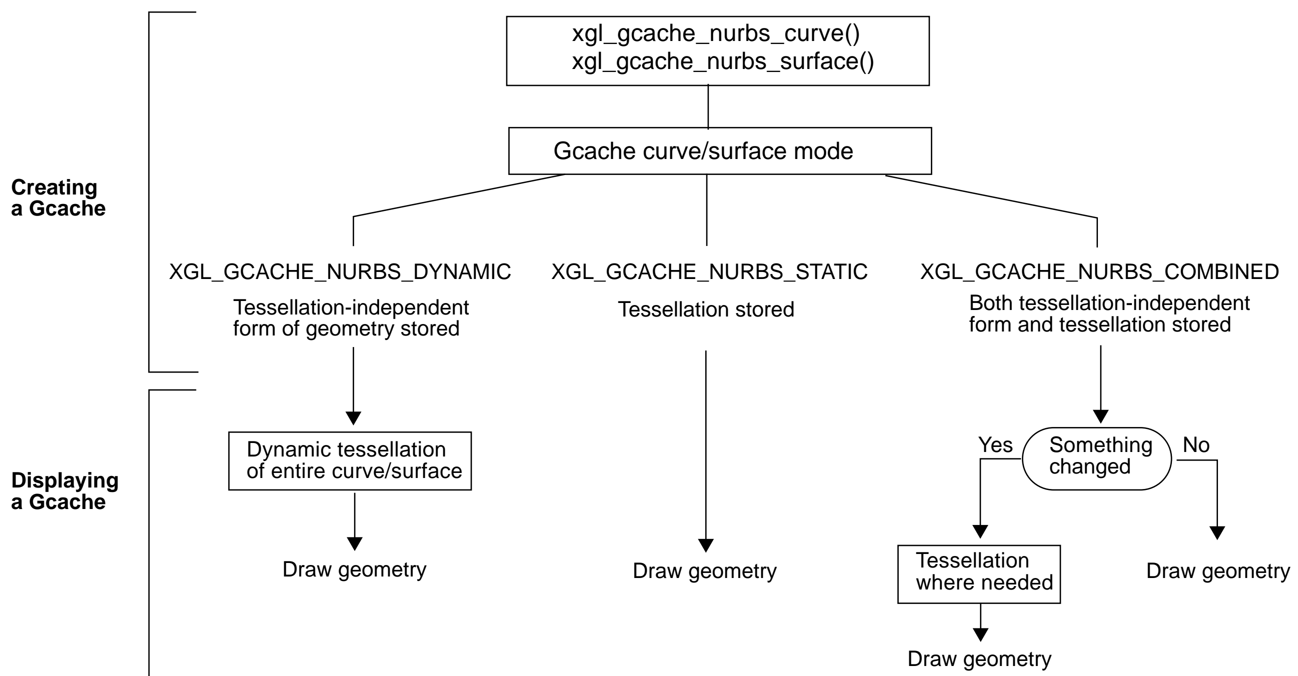


Figure 16-1 Gcache NURBS Curve and Surface Representation Modes

Displaying the Gcache

Usually an application will use one Gcache object for each primitive that needs to be cached. The primitive is rendered by calling `xgl_context_display_cache()`. This operator is used to display the Gcache on a specified device and to compare the saved state (attribute values) of the Gcache with the current state in the Context, returning the result of this comparison. This allows the application to decide if it will display the Gcache when the Gcache state and the related Context attributes differ.

The operator is defined as:

```
Xgl_cache_display
xgl_context_display_gcache(
    Xgl_ctx             ctx,
    Xgl_gcache          gcache,
    Xgl_boolean         test,
    Xgl_boolean         display);
```

The parameter `test` controls whether the state in the Gcache is validated. If `test` is `TRUE`, the saved state is compared with the current state, and `xgl_context_display_gcache()` will return either `XGL_CACHE_DISPLAY_OK` or `XGL_CACHE_ATTR_STATE_DIFFERENT`. If `test` is `FALSE`, then `xgl_context_display_gcache()` returns `XGL_GCACHE_NOT_CHECKED`.

The parameter `display` controls whether the Gcache is rendered on the device associated with the Context. If `display` is `FALSE`, the Gcache is not rendered. If `display` is `TRUE`, and `test` is `FALSE`, the Gcache is rendered. If both `display` and `test` are `TRUE`, the Gcache is only rendered if the validity test is `XGL_CACHE_DISPLAY_OK`.

Gcache Example Programs

The following example programs show a non-self-intersecting polygon being rendered with and without a Gcache, and a complex polygon being rendered with and without a Gcache. To compile the programs, type `make gcache` in the example programs directory. The complete program includes the `ex_utils.c` utility file listed in Appendix B, `gcache_main.c` listed in Appendix B, and the two Gcache example programs listed in this chapter.

Gcache of Simple Polygon

This program displays a non-self-intersecting polygon. Figure 16-2 shows the program output.

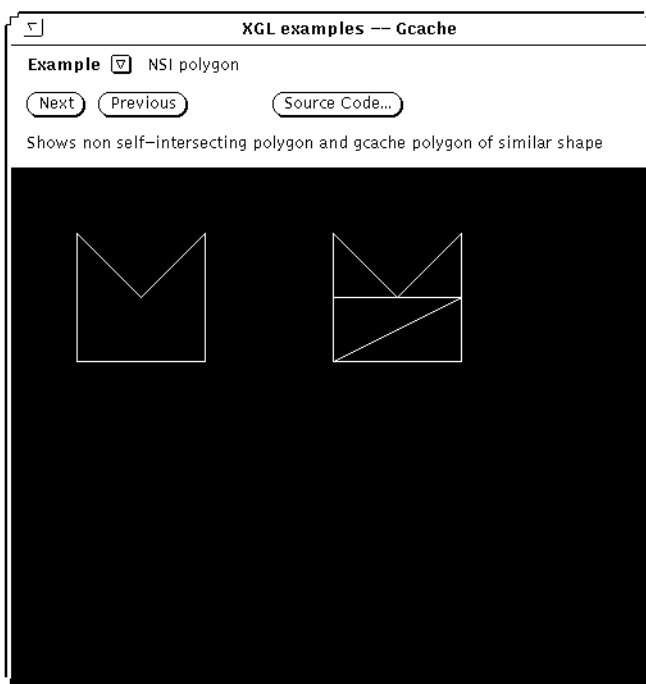


Figure 16-2 Output of `gcache_nsi_pgon.c`

Code Example 16-1 Gcache of Simple Polygon

```
/*
 * gcache_nsi_pgon.c
 */
#include "ex.h"

/*
 * Compare a non self-intersecting polygon with a gcache polygon
 * of similar shape
 */

gcache_nsi_pgon(ctx)
    Xgl_ctx   ctx;
{
    Xgl_pt_list       pl[2];
    static Xgl_pt_f3d pts_f3d[2][5] =
```

```
            {{{50, 50, 0}, {100, 100, 0}, {150, 50, 0},
            {150, 150, 0}, {50, 150, 0}}, {{250, 50, 0},
            {300, 100, 0}, {350, 50, 0}, {350, 150, 0},
            {250, 150, 0}}};
    Xgl_object          gcache;
    int                 i;
    Xgl_color           surface_color;
    Xgl_color           edge_color;

    /* Prepare data for polygon and gcache polygon */
    for (i = 0; i < 2; i++) {
        pl[i].pt_type = XGL_PT_F3D;
        pl[i].bbox = NULL;
        pl[i].num_pts = 5;
        pl[i].pts.f3d = pts_f3d[i];
    }

    xgl_context_new_frame(ctx);
    /* set surface & edge colors, turn on edges */
    surface_color = blue_color;
    edge_color = yellow_color;
    xgl_object_set(ctx,
               XGL_CTX_SURF_EDGE_FLAG, TRUE,
               XGL_CTX_EDGE_COLOR, &edge_color,
               XGL_CTX_SURF_FRONT_COLOR, &surface_color,
               NULL);



    /*
     * The polygon is drawn on the left and the gcache polygon on
     * the right
     */
    xgl_polygon(ctx, XGL_FACET_NONE, NULL, NULL, 1, &pl[0]);

    gcache = xgl_object_create(sys_st,
                 XGL_GCACHE, NULL,
                 XGL_GCACHE_DO_POLYGON_DECOMP, TRUE,
                 XGL_GCACHE_SHOW_DECOMP_EDGES, TRUE,
                 XGL_GCACHE_POLYGON_TYPE, XGL_POLYGON_NSI,
                 NULL);
    xgl_gcache_polygon(gcache, ctx, XGL_FACET_NONE, NULL, NULL, 1,
                  &pl[1]);
    xgl_context_display_gcache(ctx, gcache, FALSE, TRUE);
    xgl_context_post(ctx, TRUE);
}
```

Gcache of Complex Polygon

The following example program displays a complex self-intersecting polygon. Figure 16-3 shows the program output.

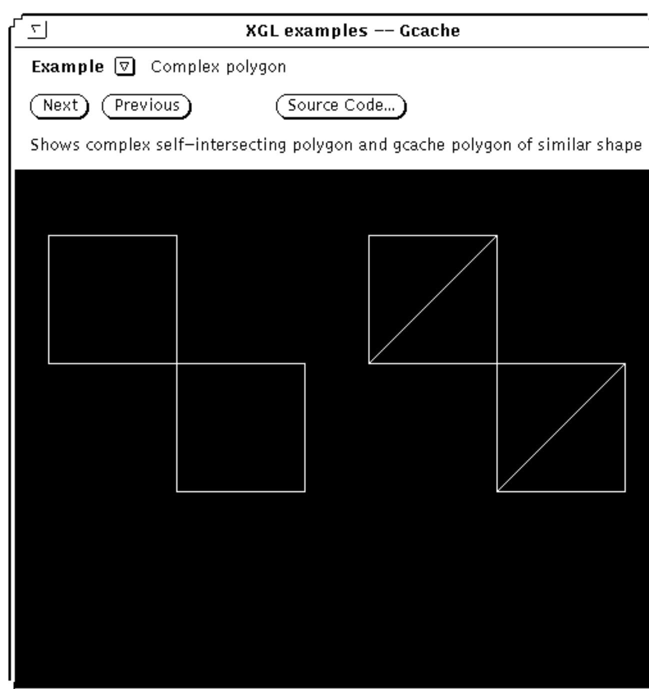


Figure 16-3 *Output* of `gcache_complex_pgon.c`

Code Example 16-2 Gcache of Complex Polygon

```
/*
 * gcache_complex_pgon.c
 */

#include "ex.h"

/* Compare a complex polygon with a gcache polygon of similar
 * shape */

gcache_complex_pgon(Xgl_objectctx)
```

```
{
    Xgl_pt_list          pl[2];
    static Xgl_pt_f3d    pts_f3d[2][6] =
              {{{25, 50, 0}, {125, 50, 0}, {125, 250, 0},
              {225, 250, 0}, {225, 150, 0}, {25, 150, 0}},
              {{275, 50, 0}, {375, 50, 0}, {375, 250, 0},
              {475, 250, 0}, {475, 150, 0}, {275, 150, 0}}};
    Xgl_object    gcache;
    int           i;
    Xgl_color     surface_color;
    Xgl_color     edge_color;

    /* Prepare data for polygon and gcache polygon */
    for (i = 0; i < 2; i++) {
          pl[i].pt_type = XGL_PT_F3D;
          pl[i].bbox = NULL;
          pl[i].num_pts = 6;
          pl[i].pts.f3d = pts_f3d[i];
    }

    xgl_context_new_frame(ctx);
    /* set surface & edge colors, turn on edges */
    surface_color = blue_color;
    edge_color = yellow_color;
    xgl_object_set(ctx,
              XGL_CTX_SURF_EDGE_FLAG, TRUE,
              XGL_CTX_EDGE_COLOR, &edge_color,
              XGL_CTX_SURF_FRONT_COLOR, &surface_color,
              NULL);

    /*  The polygon is drawn on the left and the gcache polygon on
     *  the right */
    xgl_polygon(ctx, XGL_FACET_NONE, NULL, NULL, 1, &pl[0]);

    gcache = xgl_object_create(sys_st,
                     XGL_GCACHE, NULL,
                     XGL_GCACHE_DO_POLYGON_DECOMP, TRUE,
                     XGL_GCACHE_SHOW_DECOMP_EDGES, TRUE,
                     XGL_GCACHE_POLYGON_TYPE,
                     XGL_POLYGON_COMPLEX,
                     NULL);
    xgl_gcache_polygon(gcache, ctx, XGL_FACET_NONE, NULL, NULL, 1,
                      &pl[1]);
    xgl_context_display_gcache(ctx, gcache, FALSE, TRUE);
    xgl_context_post(ctx, TRUE);
}
```

Texture Mapping 17

This chapter describes the XGL texture mapping capabilities. The chapter includes information on the following topics:

- How a texture image is created and mapped to a surface
- Color interpolation and composition
- Attributes and data structures used in texture mapping

Overview of Texture Mapping

Texture mapping in computer graphics generally refers to the process of mapping a two-dimensional image onto geometric primitives. The primitives are annotated with an extra set of 2D coordinates that orient the image on the primitive. The coordinate system axes of the image space are typically denoted *u* and *v* for the horizontal and vertical axes, respectively. When the geometry is processed, the texture is applied to the geometry and appears draped over the geometry definition like paint or cloth.

The texture to be draped on the geometric primitive can be stored as an array of colors that will eventually be mapped onto the polygonal surface. The surface to be textured is specified with vertex coordinates and texture coordinates (*u*,*v*), the latter being used to map the color array on the polygon's surface. The *u* and *v* are interpolated across the span and then used as indices into the texture map to obtain the texture color. This color is combined with the

primitive color (obtained by interpolating vertex colors across spans) or the colors specified by the application to obtain a final color value at the pixel location.

Texture maps do not have to be color arrays; they can also be arrays of intensities used for color modulation. In this case, the application can specify two colors to modulate with the intensity, or it can take one of the colors from the primitive. The software takes the colors and uses the intensity in the texture map to determine how much of each color to blend to produce the color of the pixel. This is useful for defining mottled textures found in landscape or cloth.

Texture maps can provide reasonable images in only very constrained situations. For a more robust implementation, MIP maps are required.

MIP Maps

The ideal projection of pixel areas results in a computationally expensive process. This process may involve filtering the source image texture values (texels) over arbitrarily shaped large areas. A MIP map approach provides an approximation to an ideal filtering of an image. In MIP mapping, a series of filters is applied to the original texture image. Each successive filter results in successively smaller, filtered versions of the orginal texture. Each of these filtered images constitutes a layer of the MIP map pyramid.

Because the MIP map pyramid represents a hierarchy of filters applied to a single source image, the visual quality of a particular texture is strongly related to the characteristics of the filters used. The XGL library supports several methods for combining samples from a pyramid to produce an output sample, such as point sampling, bilinear interpolation on a single level, or trilinear interpolation between two levels.

Multiple Textures

Using XGL, the application can apply several textures to an individual primitive while rendering. This capability is important for certain images. For example, in rendering a helicopter, an application could texture the sides with camouflage painting and decal text over the camouflage. This is done by applying two textures: a first texture for the camouflage, and a second texture for the text. The second texture uses α values to make the text transparent

everywhere except where the letters appear. The application can also specify the stage in the rendering process where a texture should apply, for example, before or after lighting is applied, or after depth cueing is applied.

How Texture Mapping Works in XGL

XGL provides texture mapping functionality via the MipMap Texture object and the Texture Map object. The MipMap Texture object encapsulates texture images and provides the filtering capabilities to generate a MIP map. The Texture Map object references a texturing description structure that specifies how textures are applied to a primitive.

Note – At this release, a new object, the Texture Map object, has been provided for texture mapping. The Data Map Texture object is retained for backward compatibility, but it will be removed from the XGL library at a future release. It is recommended that application programs use the Texture Map object.

Overview Steps for Mapping a Texture to a Surface

You map a texture to a surface by completing these general procedures.

1. Define the texture data by creating a texture image and writing the image data to one or more Memory Raster objects. See "Defining the Texture Data" on page 454.
2. Specify the properties of the texture in the Texture Map object. Note that unexpected results may occur if you mix the use of the Texture Map object and the Data Map Texture object in the same Context. See "Specifying the Properties of the Texture" on page 458.
3. Provide texture coordinates in the vertex data of a primitive. The texture coordinates are used to orient a texture map on the primitive. See "Mapping the Texture to a Surface Primitive" on page 481

For an example of a texture mapping program, see page 484.

Note – Texturing is only supported when surfaces are rendered through a 3D Context object and when the Raster attached to the 3D Context has a color type of `XGL_COLOR_RGB`.

Defining the Texture Data

In XGL, texture image data resides in a MIP map pyramid. A MIP map pyramid stores an array of prefiltered versions of the texture image. Each image in the array has half the dimension of the image before it. The MIP map pyramid typically consists of a series of levels, with each level containing an image. The MIP map data is encapsulated in a MipMap Texture object, which is used by a texture object when rendering textured primitives.

Before the texture can be stored in a MipMap Texture object, it must be written into an XGL Memory Raster object. There are two ways for an application to provide texture data:

- Store the image in a Memory Raster object and then use an XGL utility operator to create the MIP map. The utility creates the MIP map pyramid and encapsulates it in a MipMap Texture object.
- Provide an application-defined MIP map and store the levels of the MIP map in an array of Memory Rasters, with one level of the MIP map in each Memory Raster.

The sections below provide information on defining the texture data in these ways.

Letting XGL Build the Mip Map

XGL automatically creates a MIP map from a texture pattern in a Memory Raster object. To generate a MIP map in this way, follow these procedures:

1. Create a Memory Raster object and write the texture pattern into it. See “Writing the Texture Data into a Memory Raster” on page 455.
2. Create a MipMap Texture object. See “Creating a MipMap Texture Object” on page 456.
3. Create a MIP map using the MipMap Texture operator `xgl_mipmap_texture_build()` to automatically generate the MIP map pyramid from the Memory Raster. “Converting the Texture Data to MIP Map Format” on page 457.

Writing the Texture Data into a Memory Raster

A texture image consists of an array of values for each texel. These values can provide red, green, and blue color values, an α value, or a scale factor. Each 8 bits of data provides one piece of information, such as the red component of the color at the pixel; this piece of information is referred to as a channel. A texture image can have from one to four channels.

The texture for a texturing operation is initially written into an XGL Memory Raster object. The Memory Raster for a MipMap object can be 8-bit or 32-bit, where a Memory Raster with a depth of 8 bits is used to create a one-channel texture image, and a Memory Raster with a depth of 32 bits is used to create a four-channel texture image. The application can create a 24-bit texture image by creating a 32-bit Memory Raster and setting the depth to 24; in this case, the extra 8 bits of the Memory Raster are ignored. A Memory Raster object with four channels is created as follows.

```
mem_ras = xgl_object_create(sys_state, XGL_MEM_RAS, NULL,
                            XGL_DEV_COLOR_TYPE, XGL_COLOR_RGB,
                            XGL_RAS_DEPTH, 32,
                            XGL_RAS_HEIGHT, 8,
                            XGL_RAS_WIDTH, 8, NULL);
```

The values written into a Memory Raster are unsigned, and each channel has a value between 0 and 255. During a texturing operation, the value in each channel is divided by 255 to obtain a floating point value, and this value is then used in the texturing computation. In a four-channel texture map, the color values are stored in RGBα format, that is, the uppermost byte is α, followed by 8 bits each of blue, green, and red intensity values.

Note – The color type of the Memory Raster used in the texture image creation is ignored during texturing operations. Thus, the values in the Memory Raster are used as is, and no color conversion is performed.

Once the Memory Raster object is created, its address is retrieved, and the texture pattern is written into memory. To get the address of a Memory Raster, use the attribute `XGL_MEM_RAS_IMAGE_BUFFER_ADDR`, as shown in the code fragment that follows.

```
/* Create a mem ras object; 8x8 32-bits deep */
 mem_ras_front = xgl_object_create(sys_st, XGL_MEM_RAS, NULL,
                                 XGL_DEV_COLOR_TYPE, XGL_COLOR_RGB,
                                  XGL_RAS_DEPTH, depth,
                                  XGL_RAS_WIDTH, width,
                                  XGL_RAS_HEIGHT, height, NULL);

 /* get memory address of the raster */
 xgl_object_get(mem_ras_front, XGL_MEM_RAS_MEMORY_ADDRESS,
                &addr);

 /* write texture into memory */
 tmp_addr = addr;

 /* Width is a multiple of 2 for this example */
 /* Write a checked board pattern into memory */
 for (i = 0; i < height; i += 2) {
    for (j = 0; j < width; j+= 2) {
          *tmp_addr++ =  0xFFFFFFFF;
          *tmp_addr++ = 0x00000000;
     }
     for (j = 0; j < width; j+= 2) {
          *tmp_addr++ =  0x00000000;
          *tmp_addr++ = 0xFFFFFFFF;
     }
 }
```

For more information on the memory representation of a Memory Raster, see the reference manual page for the `XGL_MEM_RAS_IMAGE_BUFFER_ADDR` attribute. For information on Memory Raster objects and attributes, see also Chapter 4, “Devices”.

Creating a MipMap Texture Object

To create a MipMap Texture object, use `xgl_object_create()` as follows:

```
mipmap = xgl_object_create(sys_state, XGL_MIPMAP_TEXTURE, NULL, 0);
```

Converting the Texture Data to MIP Map Format

To request that the MipMap Texture object generate the MIP map, use the operator `xgl_mipmap_texture_build()`. This operator takes as input a single Memory Raster, which is used as the base texture to compute an entire MIP map pyramid of Memory Rasters on behalf of the application. XGL uses the Mitchell filter to create the MIP map.

The application first must define the number of levels for the MIP map. To compute the number of levels for a square Memory Raster, take the logarithm to the base of 2 of the width or height. If the Memory Raster is not square, take the logarithm to the base of 2 of the smaller of the width or height. Set the attribute `XGL_MIPMAP_TEXTURE_LEVELS` to the value.

Once the number of levels has been defined, the application can call the `xgl_mipmap_texture_build()` operator. This operator is defined as:

```
xgl_mipmap_texture_build(mipmap, in_mem_ras, u_bound, v_bound);
```

where:

`mipmap`	A MipMap Texture object
`in_mem_ras`	The Memory Raster object that the application wants to turn into a MIP map within the MipMap Texture object
`u_bound, v_bound`	Specify how the application wants to handle texture borders. These hints are used during the MIP map creation. The values for `u_bound`, `v_bound` are listed in Table 17-1.

Table 17-1 Texture Border Conditions

Value	Description
XGL_TEXTURE_BOUNDARY_WRAP	The texture is repeated across the face of a polygon, and the border between each copy should appear seamless.
XGL_TEXTURE_BOUNDARY_TRANSPARENT	The texture will be used so that it doesn't cover the entire face of a polygon, and outside its boundaries it will be transparent. This option could also be used if the texture were going to be wrapped and the border between each copy does not need to appear seamless.
XGL_TEXTURE_BOUNDARY_CLAMP	The texture will be used so that it doesn't cover the entire face of a polygon, and outside its boundaries it will clamp to the values given by the application in the `boundary_values` field of the *Xgl_texture_mipmap_desc* structure.
XGL_TEXTURE_BOUNDARY_MIRROR	Texel sampling is reversed across the texture map, creating a mirrored alternating pattern of the texture map across the primitive.
XGL_TEXTURE_BOUNDARY_CLAMP_BOUNDARY	The texture will be used so that it doesn't cover the entire face of a polygon, and outside its boundaries it will clamp to a color specified by the closest boundary texel.

The Memory Raster data is moved to a device-independent MIP map format internal to the MipMap Texture object; thus, after the `xgl_mipmap_texture_build()` operator has been called, the original Memory Raster can be deleted.

Creating a MipMap Texture object and calling `xgl_mipmap_texture_build()` is shown in the example below.

```
/* Create a front MipMap Texture object */
    front_tm = xgl_object_create(sys_st, XGL_MIPMAP_TEXTURE,
                                 NULL, 0);

/* Do the filtering operation on the front mipmap */
/* Set the number of levels for the front MipMap Texture object */
    xgl_object_set(front_tm, XGL_MIPMAP_TEXTURE_LEVELS, 3, NULL);
```

```
/* Set the boundary conditions for generating the MIP map */
    u_bound = XGL_TEXTURE_BOUNDARY_WRAP;
    v_bound = XGL_TEXTURE_BOUNDARY_WRAP;

/* Build  a MIP map pyramid of level 3 from the mem ras */
xgl_mipmap_texture_build(front_tm, in_mem_ras, u_bound, v_bound);
```

Using an Application-Supplied MIP Map

The application can create its own MIP map data, generating an array of Memory Rasters with the filtered data needed for each MIP map level. In this case, the application uses its own methods of filtering and stores the MIP map pyramid in a list of Memory Rasters using standard ways of creating and drawing into Memory Rasters. It then passes this set of Memory Rasters to the MipMap Texture object. The dimension of each Memory Raster in the list should have half the dimension of the previous Memory Raster.

When the application has stored its MIP map pyramid in XGL's Memory Raster format, it must set the number of MIP map levels with the attribute `XGL_MIPMAP_TEXTURE_LEVELS`. The number of Memory Rasters in the list equals the number of levels.

When the number of levels has been specified, the application passes the data in the Memory Rasters to the MipMap Texture object using the `XGL_MIPMAP_TEXTURE_MEM_RAS_LIST` attribute and `xgl_object_set()`, as in the example below. The `XGL_MIPMAP_TEXTURE_MEM_RAS_LIST` attribute specifies an array of Memory Rasters that form a MIP map pyramid. Each level of the pyramid has dimensions that are half as large as the previous level.

```
xgl_object_set(mipmap,
           XGL_MIPMAP_TEXTURE_MEM_RAS_LIST, memras_ptr_list,
           NULL);
```

This call copies the application MIP maps into a device-independent format internal to XGL. When the data has been copied, the original Memory Rasters can be deleted.

Determining the Size of the Base Texture Image

To determine the size of the base texture image, use the read-only attributes XGL_MIPMAP_TEXTURE_WIDTH and XGL_MIPMAP_TEXTURE_HEIGHT. These values are set indirectly when the application uses the operator xgl_mipmap_texture_build() or sets the attribute XGL_MIPMAP_TEXTURE_MEM_RAS_LIST. The XGL_MIPMAP_TEXTURE_DEPTH attribute defines the number of bits used to specify one texture element. The number of channels per texture element is obtained by dividing the depth of the texture by 8.

Specifying the Properties of the Texture

There are two objects that the application can use to associate texture mapping information with a 3D Context: the Texture Map object and the Data Map Texture object. Each object references the image data in a MipMap Texture object for the texturing data. The differences between the two texture objects are as follows:

- Texture Map object – Has a one-to-one correspondence with a MipMap object. This is the recommended interface for this release, and it will be the only supported interface for future releases.
- Data Map Texture object – Can reference multiple textures in an array of MipMap objects. This interface is supported at this release but may not be supported in future releases.

The general procedure for setting up these objects is as follows:

1. Create a texture object of either type.

Note – Do not use Texture Map objects and Data Map Texture objects with the same Context, or unexpected results may occur. Use Texture Map objects unless the application has already been written with Data Map Texture objects.

---

2. Each object has a texturing description structure that defines the MipMap Texture object(s) to be used, specifies how texel color is obtained, and describes boundary conditions for the texture. Fill in the texturing description structure with the appropriate control parameters and associate the structure with the object.

3. Associate the texture object with a 3D Context.

The following sections provide information on setting up these objects and their associated data structures.

Creating a Texture Map Object

A Texture Map object is created with `xgl_object_create()`:

```
texture_map = xgl_object_create(sys_state, XGL_TMAP,
                                NULL, NULL);
```

An application defines the texturing control parameters and the texture image (defined in the MipMap Texture object) in a texture descriptor structure. The descriptor structure is associated with the Texture Map object using the `XGL_TMAP_DESCRIPTOR` attribute, as shown below.

```
xgl_object_set(tmap, XGL_TMAP_DESCRIPTOR, &tm_desc, 0);
```

Specifying Texture Properties With the Texture Map Object

The texture descriptor structure for the Texture Map object is *Xgl_texture_general_desc:*

```
typedef struct {
    Xgl_texture_type                    texture_type;
    Xgl_texture_comp_info               comp_info;
    union {
        Xgl_texture_general_mipmap_desc mipmap;
        Xgl_usgn32                      unused[64];
    } texture_info;
} Xgl_texture_general_desc;
```

This structure specifies the type of texture map that the application will use, specifies composition information, and provides a description of the MIP map. These fields are discussed in more detail in the following sections.

Texture Type

The texture descriptor defines the type of the texture map. Currently, the only value for texture type is XGL_TEXTURE_TYPE_MIPMAP.

Composition Method

The texture descriptor points to a structure that specifies the composition method. The composition method specifies how the texture map is blended with the primitive's existing data. At this release, the only possible composition method is color composition.

```
typedef union {
    Xgl_texture_color_comp_info      color_info;
} Xgl_texture_comp_info;
```

Color Composition

The color composition method defines how the information present in the texture image is used to compose the final color value. The Texture Map object provides several color composition methods, such as blend, decal and modulate modes. The application specifies this using the *Xgl_texture_color_comp_info* structure. This structure is defined as:

```
typedef struct {
    Xgl_usgn32                   num_render_comp_desc;
    Xgl_render_component_desc    render_component_desc[2];
    Xgl_usgn32                   num_channels[2];
    Xgl_usgn32                   channel_number[2];
} Xgl_texture_color_comp_info;
```

The argument num_render_comp_desc specifies the number of rendering component descriptors used by the Texture Map object. The rendering component descriptor render_component_desc defines where in the pipeline a texture map is applied. The first element of the render_component_desc array is used in composing the color, and the second element is used in composing the alpha value. The num_channels array specifies how many channels in the texture are used to modify the destination color. Like the render_component_desc array, the first element

of the num_channels array is used in composing the color, and the second element is used in composing the alpha value. The value of the num_channels element can be either 3 or 1. If the value of num_channels is 1, the corresponding element in the channel_number field determines the channel number in the texture image that is used in composing the color. For example, if num_channels[0] is 1 and if channel_number[0] is 0, channel number 0 in the texture image is used in composing color. Note that since alpha uses only one channel, num_channels[1] is always 1; however, channel_number[1] specifies which channel in the texture image is used in composing alpha. Note also that channel_number is ignored if num_channels is set to 3.

Rendering Component Descriptor for Final Texel Color

The field render_component_desc describes the stage of the rendering pipeline that is affected by the texture descriptor and describes how the color intensity from the texture image (MipMap Texture object) defined in the texture descriptor is combined to obtain the final color value. The Xgl_render_component_desc structure is defined as:

```
typedef struct {
    Xgl_render_component                    comp;
    Xgl_texture_op                          texture_op;
    union Xgl_op_info {
        Xgl_texture_blend_op                blend;
        Xgl_texture_decal_op                decal;
        Xgl_texture_blend_intrinsic_op      blend_int;
    } op;
} Xgl_render_component_desc;
```

The comp parameter defines the stage in the rendering pipeline that is affected by the texture description. When multiple textures are applied to a single surface, they are applied to the rendering stage designated in the comp field at a time that is most convenient for the renderer. If more than one texture is to apply to the same component, they are applied in the order that their texture descriptors are listed in the Texture Map object. Note that since alpha values are unaffected by lighting or depth cueing, the comp parameter is only valid for color composition and is ignored for render_component_desc[1].

Table 17-2 shows the possible values for the `comp` parameter.

Table 17-2 Texture Mapping in the Rendering Pipeline

Value	Description
`XGL_RENDER_COMP_DIFFUSE_COLOR`	This value designates the diffuse or intrinsic color of the surface as the component to apply the texture to. When textures are applied to this component, the surface can also have lighting applied.
`XGL_RENDER_COMP_REFLECTED_COLOR`	This value designates the color obtained after lighting is modified using texturing.
`XGL_RENDER_COMP_FINAL_COLOR`	This value indicates to XGL that the final resulting color (after lighting and depth cueing) will be modified by a color from the texture.

The `texture_op` parameter defines how the texture map is combined with the current color or alpha value to obtain a final color. Table 17-3 lists the available texture mapping compositing methods. See the `XGL_TMAP_TEXTURE_DESCRIPTOR` man page for mathematical definitions of the methods.

Table 17-3 Texture Mapping Compositing Methods

Value	Description
`XGL_TEXTURE_OP_MODULATE`	Texture values modulate the object color or alpha value.
`XGL_TEXTURE_OP_DECAL`	Decal mode is only available when a four-channel texture map is used. In this mode, the alpha channel of the texture map specifies how to blend the texture color with a constant color, resulting in the texture appearing like a decal on the object.
`XGL_TEXTURE_OP_DECAL_INTRINSIC`	Decal mode is only available when a four-channel texture map is used. In this mode, the alpha channel of the texture map specifies how to blend the texture color with the object color.
`XGL_TEXTURE_OP_BLEND`	The object color is replaced by two constant colors modulated by the texture color. For example, you could define mountains with grassy areas and trees using a blend between brown and green. In the case of alpha, the incoming alpha value is modulated by the texture alpha to produce the final alpha value.

Table 17-3 Texture Mapping Compositing Methods *(Continued)*

Value	Description
XGL_TEXTURE_OP_BLEND_INTRINSIC	The object color and a constant color are blended by the texture map color to produce the destination color. In the case of alpha, the incoming alpha value is modulated by the texture alpha to produce the final alpha value.
XGL_TEXTURE_OP_REPLACE	The texture value replaces the object color, or the texture image alpha value replaces the the alpha value. This mode is only supported when num_channel[0] is 3.

The op parameter in the *Xgl_render_component_desc* provides the constant color for the XGL_TEXTURE_OP_DECAL, XGL_TEXTURE_OP_BLEND, or XGL_TEXTURE_OP_BLEND_INTRINSIC texture mapping compositing methods.

Channel Information

For the compositing methods XGL_TEXTURE_OP_MODULATE, XGL_TEXTURE_OP_BLEND, and XGL_TEXTURE_OP_BLEND_INTRINSIC, channel information can be used as a way to pack information. For color values, the value of num_channels[0] specifies how many channels in the texture are used to modify the destination color. To use a single channel of information to modify the red, green, and blue destination colors, set the num_channels[0] parameter to 1, and set channel_number to the channel in the texture to be used. To use three channels of information, with one channel for each of red, green, and blue destination colors, set num_channels[0] to 3; in this case, channel_number is ignored.

The value of num_channels[1] is used in composing the alpha value. Since alpha uses only one channel, num_channels[1] is always 1; however, the value of channel_number[1] specifies which channel in the texture image should be used in composing alpha. The channel number can be any of the four channels.

MIP Map Descriptor

The MIP map to be used as the texture in a texture descriptor structure is defined by an *Xgl_texture_general_mipmap_desc* structure.

```
typedef struct {
    Xgl_texture                  texture_map;
    float                        max_u_freq;
    float                        max_v_freq;
    Xgl_texture_boundary         u_boundary;
    Xgl_texture_boundary         v_boundary;
    Xgl_usgn8                    boundary_values[4];
    float                        depth_interp_factor;
    Xgl_texture_interp_info      interp_info;
    Xgl_matrix_f2d               orientation_matrix;
} Xgl_texture_general_mipmap_desc;
```

Pointer to MipMap Texture Object

The `texture_map` field specifies the MipMap Texture object that contains the texture image.

Note – At this release, the fields `max_u_freq` and `max_v_freq` are not implemented.

Boundary Conditions

The `u_boundary`, `v_boundary` fields of the *Xgl_texture_mipmap_desc* structure specify how to apply texturing to a primitive when the texture bounds are exceeded. Typically when texturing a polygon, the colors that are used come from the texturing process. However, it is possible to texture a polygon in such a way that the texture does not cover the entire face of the polygon (by specifying *u* and *v* values with the primitive's vertices that are less than 0.0 or greater than 1.0). When this happens, the user can specify whether to wrap (repeat) or mirror (reverse) the texture again and again until the polygon is completely covered, or he can specify a color or intensity to clamp to. Clamping results in a polygon with one copy of the texture and the remaining part of the polygon filled in with the color of a constant clamp value or the boundary texel. He may also specify that the texture becomes transparent once

the edge of the texture bounds are reached, which means that the remaining untextured part of the polygon takes on the appearance that it would have had if the current texture had not been applied.

The `u_boundary`, `v_boundary` fields are flags that indicate how the renderer should finish rendering the polygon when the edge of the texture has been reached. Table 17-4 lists possible values for these fields.

Table 17-4 MipMap Descriptor Boundary Conditions

Value	Description
XGL_TEXTURE_BOUNDARY_CLAMP	Finish texturing using a constant value.
XGL_TEXTURE_BOUNDARY_WRAP	Repeat the texture.
XGL_TEXTURE_BOUNDARY_TRANSPARENT	Finish rendering as though texturing were not being done.
XGL_TEXTURE_BOUNDARY_MIRROR	Reverse the texture.
XGL_TEXTURE_BOUNDARY_CLAMP_BOUNDARY	Finish texturing using a color specified by the closest boundary texel.

Note – If the *u* and *v* boundary conditions are not identical (for example, if one specifies clamp and the other specifies transparent), and if both the *u* and *v* values exceed the texture boundary in both directions, then an ambiguous situation arises where the renderer can't determine which method to use. In this case, the renderer follows the following order of precedence in deciding which of the boundary methods to use:

1. XGL_TEXTURE_BOUNDARY_TRANSPARENT
2. XGL_TEXTURE_BOUNDARY_CLAMP
3. XGL_TEXTURE_BOUNDARY_WRAP
4. XGL_TEXTURE_BOUNDARY_MIRROR
5. XGL_TEXTURE_BOUNDARY_CLAMP_BOUNDARY

Boundary Condition Clamp Color Source

The `boundary_values` field specifies the value (color) to clamp to when either the *u* or *v* value exceeds the texture boundaries. This is the value used by the `XGL_TEXTURE_BOUNDARY_CLAMP` value of the boundary condition fields. The array index 0 corresponds to red, the index 1 to green, the index 2 to blue, and the index 3 to α.

Depth Sampling

The `depth_interp_factor` field modifies the calculation of which MIP map level to sample. There are times when the application would prefer that the texture be more or less aliased as it is applied to a surface (more jaggy or more blurry). This can be accomplished by changing the texture map depth level at which texels are sampled. This attribute is a floating point number that allows the application to adjust the sample depth level in either direction. A value of 1.0 moves away from the MIP map base towards more blurry levels. A value of -1.0 moves the depth towards the MIP map pyramid's base, which has more detail in it, and may make the picture appear more jaggy.

Texture Sampling Method

During the texturing process, the data from the texture MIP map can be sampled or interpolated in a number of ways in order to come up with a value to apply to the pixel. The application specifies how to sample and interpolate the MIP map data using the *Xgl_texture_interp_info* structure, which is defined as:

```
typedef struct {
    Xgl_texture_interp_method     filter1;
    Xgl_texture_interp_method     filter2;
} Xgl_texture_interp_info;
```

`filter1` is applied when the pixel being textured maps to an area greater than one texel. `filter2` is applied when the pixel being textured maps to an area less than or equal to one texel. The values for the *Xgl_texture_interp_method* structure are the texture sampling methods listed in Table 17-5.

Table 17-5 Texture Sampling Methods

Value	Description
`XGL_TEXTURE_INTERP_POINT`	Sample one texel point from the base top or first MIP map level at the closest *u* and *v* to the pixel.
`XGL_TEXTURE_INTERP_BILINEAR`	Sample the four texels closest to the center of the pixel from the base MIP map level and average them.
`XGL_TEXTURE_INTERP_MIPMAP_POINT`	Sample one texel point from the nearest MIP map level at the closest *u* and *v* to the pixel.

Table 17-5 Texture Sampling Methods

Value	Description
XGL_TEXTURE_INTERP_MIPMAP_BILINEAR	Sample the four texels closest to the center of the pixel from the nearest MIP map level and average them.
XGL_TEXTURE_INTERP_MIPMAP_TRILINEAR	Sample the four texels closest to the center of the pixel from the nearest MIP map level above the depth *d* and the four texels closest to the MIP map level below the depth *d* and average all eight of them.
XGL_TEXTURE_INTERP_MIPMAP_PT_LINEAR	Choose the two MIP maps that most closely match the size of the pixel being textured and find the nearest texel to the center of the pixel to produce a texture value from each MIP map. The final texture value is an average of those two values.

Orientation Matrix

The `orientation_matrix` field in the MipMap descriptor structure defines a 3×2 matrix with which textures can be rotated or translated across the face of a polygon by transforming the *u* and *v* values of the texture. The matrix is type float. All texture *u* and *v* values are transformed by the matrix as the texturing process takes place. Note that the *u*, *v* values in the primitive point list are subject to the `XGL_TMAP_T0_INDEX` and `XGL_TMAP_T1_INDEX` attributes before the orientation matrix is applied.

Associating the Texture Map Object With a 3D Context

To set the Texture Map object on the Context, specify the Texture Map object or the list of Texture Map objects with the `XGL_3D_CTX_SURF_FRONT_TMAP` attribute or the `XGL_3D_CTX_SURF_BACK_TMAP` attribute, as in the following example:

```
xgl_object_set(ctx, XGL_3D_CTX_SURF_FRONT_TMAP_NUM, 1,
               XGL_3D_CTX_SURF_FRONT_TMAP, &texture_map,
               NULL);
```

An application can apply multiple Texture Map objects to the same primitive or associate a different set of Texture Map objects for the front and back faces of a surface.

Specify the number of Texture Map objects in the texture map array with the attributes XGL_3D_CTX_SURF_FRONT_TMAP_NUM or XGL_3D_CTX_SURF_BACK_TMAP_NUM. The attributes XGL_3D_CTX_SURF_FRONT_TMAP_SWITCHES and XGL_3D_CTX_SURF_BACK_TMAP_SWITCHES specify a front and back array of on and off switches corresponding to the Context's array of Texture Map objects. After creating the array of switches, the application can turn on or off the use of Texture Map objects using these attributes.

Note – Texturing is only supported when the Raster associated with the 3D Context has a color type of RGB.

Additional Texture Map Functionality

The Texture Map object provides attributes that supply the functionality discussed in the following sections.

Perspective Correction of Textured Surfaces

The attribute XGL_3D_CTX_SURF_TMAP_PERSP_CORRECTION specifies the method used to compute texture coordinate values in surface interiors when a textured surface primitive is rendered. Possible values for this attribute are:

XGL_TEXTURE_PERSP_NONE
: Texture mapping coordinates are linearly interpolated without perspective correction.

XGL_TEXTURE_PERSP_PIXEL
: As the texture mapping coordinates are interpolated, their values are manipulated to account for a perspective projection.

XGL_TEXTURE_PERSP_NO_INTERP
: Texture mapping coordinates are not interpolated at all. Instead, texture mapping coordinates are used per-vertex only. This method is also known as vertex-level texture mapping.

Vertex-Level Texture Mapping

Vertex-level texture mapping can be considered a naive implementation of texture mapping. Traditional pixel-level texture mapping computes texture color at every pixel. Because of this, the software implementation is usually slow. If texture mapping is not supported in hardware, the software implementation is the only alternative.

Vertex-level texture mapping, on the other hand, computes texture color only per vertex of the primitive, and the interior is then simply Gouraud-shaded. Hardware accelerators that support fast triangle rendering can be used to perform the shading. However, if the original primitives cover large areas, the resulting image may be of poor quality.

To improve the quality of vertex-level texture mapping, an application may need to subdivide large primitives into a union of smaller subprimitives so that per-vertex sampling more accurately reflects the texture map. With NURBS surfaces, however, the application can tessellate the surface into a large number of polygons, thereby increasing texture quality.

Note the following limitations on vertex-level texture mapping:

- The mipmap level is always 0.
- Only front facing surfaces are supported.
- The value of *Xgl_render_component* in the *Xgl_render_component_desc* structure can only be `XGL_RENDER_COMP_DIFFUSE_COLOR`. This means that vertex-level texture mapping can only modify the color of a surface before lighting is applied.

Specifying the Source of Texture Coordinates

The Texture Map object attribute `XGL_TMAP_COORD_SOURCE` specifies the initial source within a primitive's vertex data of the texture coordinate. Depending on a texture map's parametrization method, this may or may not be the final texture coordinate (this initial source may be processed into the final texture coordinate). This attribute has the following values:

`XGL_TEXTURE_COORD_VERTEX`
: The source texture coordinate values are the vertex coordinate values, transformed to WC.

XGL_TEXTURE_COORD_NORMAL
The source texture coordinate values are the vertex normal values, transformed to WC. If a vertex normal is not explicitly defined with the primitive, the calculated vertex normal is used instead.

XGL_TEXTURE_COORD_DATA
The source texture coordinate's source is a member of the vertex's floating point data. (The index value indicates which member of the list.) If the primitive does not include the floating point data list for each vertex, this texture coordinate value will be set to 0. This is the default value.

If XGL_TMAP_COORD_SOURCE is set to XGL_TEXTURE_COORD_DATA, the attributes XGL_TMAP_T0_INDEX and XGL_TMAP_T1_INDEX define the indices into the vertex data array of the surface primitives. XGL_TMAP_T0_INDEX indicates that the *u* value of the data values is used as *u* texture coordinates, and XGL_TMAP_T1_INDEX indicates that the *v* value of the data values is used as *v* texture coordinates. The mapping of the texture onto the polygon face is done by listing *u* and *v* values in the data fields of vertices. Since it is possible to have several data values listed with each vertex, an application must use these attributes to specify which *u* and *v* values to use with the Texture Map object.

Specifying the Parameterization Method

The XGL_TMAP_PARAM_TYPE attribute defines the parameterization method for mapping a texture to a 3D surface. With the attribute XGL_TEXTURE_PARAM_EXPLICIT, the application specifies the mapping of the texture to the surface using the *u,v* values in the point list. All other parameterization methods provide environment texture mapping.

Environment texture mapping textures a surface as if it were reflected from a shiny object such as a sphere. The viewer position and the position of the surface determine how objects are reflected. For environment texture mapping, the attribute XGL_TMAP_PARAM_INFO provides information that is used in the generation of texture coordinates.

The XGL_TMAP_PARAM_TYPE attribute has the following values:

XGL_TEXTURE_PARAM_EXPLICIT
The mapping of texture to the surface is explicitly specified by the application by u,v values in the point list. This is the default value.

`XGL_TEXTURE_PARAM_REFLECT_SPHERE_EC`

The mapping is defined by reflection of the eye-to-vertex vector with respect to the vertex normal in Eye Coordinates. Specifically, the *u,v* values are derived as follows:

- Vertex and normal are transformed to World Coordinates. Eye Coordinates are obtained by transforming the World Coordinates by the view orientation, which is an orthogonal portion of the viewing transformation.
- The reflection vector is computed as follows:

  $s = i - 2 \times n \times (n \cdot i)$, where

  s is the reflection vector
  n is the unit vertex normal
  i is the unit eye-to-vertex vector
- The reflection vector ***s*** is then transformed by a reflection matrix, yielding vector *t*. The reflection matrix is specified using the *geom_orient_matrix* field of the *Xgl_texture_projection* structure. This structure is a part of the *Xgl_texture_param_info* structure, which is set using the `XGL_TMAP_PARAM_INFO` attribute.
- The *u, v* values are computed as follows:

$$u = 0.5 \times \left(\frac{t_x}{length + 1}\right)$$

$$v = 0.5 \times \left(\frac{t_y}{length + 1}\right),$$

$$length = \sqrt{t_x \times t_x + t_y \times t_y + (t_z + 1) \times (t_z + 1)}$$

`XGL_TEXTURE_PARAM_REFLECT_SPHERE_WC`

The mapping is defined by reflection of eye-to-vertex vector with respect to vertex normal in World Coordinates. Specifically, the u,v values are derived as follows:

- Vertex and normal are transformed to World Coordinates.
- The reflection vector is computed as follows:

  $s = 1 - 2 \times n \times (n \cdot i)$, where

s is the reflection vector
n is the unit vertex normal
i is the unit eye-to-vertex vector

- The reflection vector *s* is then transformed by a reflection matrix, yielding vector *{tx, ty, tz}*. The reflection matrix is specified using the *geom_orient_matrix* field of the *Xgl_texture_projection* structure. This structure is a part of the *Xgl_texture_param_info* structure, which is set using the XGL_TMAP_PARAM_INFO attribute.
- The *u,v* values are computed as follows:

$$u = 0.75 \text{ if } t_x = 0.0, t_z > 0.0$$

$$= 0.25 \text{ if } t_x = 0.0, t_z > 0.0$$

$$= \frac{-\arctan\left(\frac{t_z}{t_x}\right)}{2\pi} + 0.5 \quad \text{if } t_x < 0.0$$

$$= \frac{-\arctan\left(\frac{t_z}{t_x}\right)}{2\pi} + 1.0 \quad \text{if } t_x > 0.0, t_z > 0.0$$

$$= \frac{-\arctan\left(\frac{t_z}{t_x}\right)}{2\pi} + 0.0 \quad \text{if } t_x > 0.0, t_z > 0.0$$

$$v = \frac{\arcsin(t_y)}{\pi} + 0.5$$

where: $\left(\frac{-\pi}{2} \le \arctan(r) \le \frac{\pi}{2}\right)$ and $\left(\frac{-\pi}{2} \le \arcsin(r) \le \frac{\pi}{2}\right)$

XGL_TEXTURE_PARAM_LINEAR_EC

The mapping is defined by a linear projection in Eye Coodinates. The *u,v* values are derived as follows:

- The vertex is transformed into Eye Coordinates (see above), resulting in point {*sx, sy, sz*}.
- The specified two homogeneous vectors *p0* and *p1* are transformed from Eye Coordinates into Model Coordinates. Projection values are specified using the *linear* field of the *Xgl_texture_projection* structure. This structure is part of the *Xgl_texture_param_info* structure, which is set using the XGL_TMAP_PARAM_INFO attribute.
- The u,v values are computed as follows:

$$u = s_x \times p0_x + s_y \times p0_y + s_z \times p0_z + p0_w$$

$$v = s_x \times p1_x + s_y \times p1_y + s_z \times p1_z + p1_w$$

In the case of NURBS surfaces, depending on texture coordinate source and texture parametrization method, either the (u,v) coordinates of the point data, or tessellation vertex and normal of the NURBS are used for the texture coordinates.

The mapping of the texture onto the surface can be modified using the orientation matrix. See “Orientation Matrix” on page 469.

Specifying the Texture Domain

At this release, the XGL_TMAP_DOMAIN attribute specifies that the texture is to be applied to color domain. The attribute’s only value is XGL_TEXTURE_COLOR.

Creating a Data Map Texture Object

A Data Map Texture object is created as follows:

```
dmap_texture = xgl_object_create(sys_state, XGL_DMAP_TEXTURE,
                                 NULL, NULL);
```

An application defines the texturing control parameters and the texture image (MipMap Texture object) using a texture descriptor structure. A Data Map Texture object can have many such texture descriptors. These descriptor structures are associated with the Data Map Texture object using the XGL_DMAP_TEXTURE_DESCRIPTORS attribute. The number of texture descriptors is specified by the XGL_DMAP_TEXTURE_NUM_DESCRIPTORS attribute. When multiple textures are used, they are applied sequentially and are cumulative.

Specifying Texture Properties With the Data Map Texture Object

The texture descriptor structure for the Data Map Texture object is *Xgl_texture_desc*:

```
typedef struct {
    Xgl_texture_type              texture_type;
    Xgl_texture_color_comp_info   comp_info;
    Xgl_texture_interp_info       interp_info;
    union {
        Xgl_texture_mipmap_desc   mipmap;
        Xgl_usgn32                unused[64];
    } info;
} Xgl_texture_desc;
```

This structure does the following:

- Specifies the type of texture map to be used
- Specifies how the channels in the texture map affect rendering
- Specifies the method of sampling to be used
- Provides a description of the MIP map

These fields are discussed briefly in the sections below. Since much of the information provided in the fields and substructures of this structure is the same as that in the *Xgl_texture_general_desc* structure for the Texture Map object, references to preceding sections are provided, and the information is not duplicated here.

Texture Type

Each texture descriptor defines the type of the texture. This value is used when accessing the fields of the *info* union. Currently, the only texture type value is XGL_TEXTURE_TYPE_MIPMAP.

Color Composition Method

The comp_info field of the texture descriptor defines how the information present in the texture image is used to compose the final color value. The application specifies this using the *Xgl_texture_color_comp_info* structure. This structure is defined as:

```
typedef struct {
    Xgl_usgn32                  num_render_comp_desc;
    Xgl_render_component_desc   render_component_desc[2];
    Xgl_usgn32                  num_channels[2];
    Xgl_usgn32                  channel number[2];
} Xgl_texture_color_comp_info;
```

This structure defines where in the rendering pipeline a texture map is applied. Color composition functionality is the same in the Texture Map object and the Data Map Texture object. For complete information on color composition, see page 462.

Texture Sampling Method

During the texturing process, the data from the texture MIP map can be sampled or interpolated in a number of ways in order to arrive at a value to apply to the pixel. Texture sampling functionality in the two texture objects is the same, although texture sampling has been moved from the texture descriptor to the MIP map descriptor in the Texture Map object. See “Texture Sampling Method” on page 468 for information on texture sampling.

MIP Map Descriptor

The MIP map to be used as the texture in a texture descriptor structure is defined by an *Xgl_texture_mipmap_desc* structure.

```
typedef struct {
    Xgl_texture              texture_map;
    float                    max_u_freq;
    float                    max_v_freq;
    Xgl_texture_boundary     u_boundary;
    Xgl_texture_boundary     v_boundary;
    Xgl_usgn8                boundary_values[4];
    float                    depth_interp_factor;
} Xgl_texture_mipmap_desc;
```

Texture Map

The `texture_map` field specifies the MipMap Texture object that contains the texture image.

Note – At this release, the fields `max_u_freq` and `max_v_freq` are not implemented.

Boundary Conditions

The `u_boundary`, `v_boundary` fields of the *Xgl_texture_mipmap_desc* structure specify how to apply texturing to a primitive when the texture bounds are exceeded. This functionality is the same as that provided in the boundary conditions functionality in the MIP map descriptor for the Texture Map object. See page 466 for information.

Boundary Condition Clamp Color Source

The `boundary_values` field specifies the value (color) to clamp to when either the *u* or *v* value exceeds the texture boundaries. See “Boundary Condition Clamp Color Source” on page 467.

Depth Sampling

The `depth_interp_factor` field modifies the calculation of which MIP map level to sample. See “Depth Sampling” on page 468.

Associating the Data Map Texture Object With a 3D Context

In XGL, an application can have multiple Data Map Texture objects applied to the same primitive. In addition, XGL supports the capability of associating a different set of Data Map Texture objects for the front and back faces of a surface.

To set the Data Map Texture objects on the Context, specify the number of Data Map Texture objects in the array with the attributes XGL_3D_CTX_SURF_FRONT_DMAP_NUM or XGL_3D_CTX_SURF_BACK_DMAP_NUM. Then specify the list of Data Map Texture objects with the XGL_3D_CTX_SURF_FRONT_DMAP attribute or the XGL_3D_CTX_SURF_BACK_DMAP attribute, as in the following example:

```
xgl_object_set(ctx, XGL_3D_CTX_SURF_FRONT_DMAP_NUM, 1,
               XGL_3D_CTX_SURF_FRONT_DMAP, &dmap_texture,
               NULL);
```

The attributes XGL_3D_CTX_SURF_FRONT_DMAP_SWITCHES and XGL_3D_CTX_SURF_BACK_DMAP_SWITCHES specify a front and back array of on and off switches corresponding to the Context's array of Data Map Texture objects. After creating the array of switches, the application can turn on or off the use of Data Map Texture objects using these attributes.

Note – Texturing is only supported when the Raster associated with the 3D Context has a color type of RGB.

Additional Data Map Texture Attributes

The Data Map Texture object provides attributes that supply the functionality discussed in the following sections.

Orientation Matrix

The XGL_DMAP_TEXTURE_ORIENTATION_MATRIX attribute defines a 3×2 matrix with which textures can be rotated or translated across the face of a polygon by transforming the *u* and *v* values of the texture. The matrix is type float. All texture *u* and *v* values are transformed by the matrix as the texturing

process takes place. Note that the *u*, *v* values in the primitive point list are subject to the XGL_DMAP_TEXTURE_U_INDEX and XGL_DMAP_TEXTURE_V_INDEX attributes before the orientation matrix is applied.

Specifying the Source of Texture Coordinates

The attributes XGL_DMAP_TEXTURE_U_INDEX and XGL_DMAP_TEXTURE_V_INDEX define the indices into the vertex data array of the surface primitives and indicate which of the data values to use as *u* and *v* texture coordinates. The mapping of the texture onto the polygon face is done by listing *u* and *v* values in the data fields of vertices. Since it is possible to have several data values listed with each vertex, an application must use these attributes to specify which *u* and *v* values to use with the Data Map Texture object.

Specifying the Parameterization Method

The XGL_DMAP_TEXTURE_PARAM_TYPE attribute defines the parameterization method for mapping a texture to a 3D surface. For information about the values for this attribute, see "Specifying the Parameterization Method" on page 472.

In the case of NURBS surfaces, the (u,v) coordinates of the NURBS parameter space scaled to the (0,1) range are used for texture coordinates. For other surface primitives, the (u,v) coordinates must be supplied by the application.

The mapping of the texture onto the surface can be modified using the orientation matrix.

Mapping the Texture to a Surface Primitive

Texture is mapped onto a surface primitive via the vertex information in the primitive's point data. The point data type used for a texture-mapped surface varies for the Texture Map object and the Data Map Texture object as follows:

- Texture Map object – The point type can be any 3D point type, and the vertex values, vertex normal values, or the vertex floating point data can be used for the texture coordinates. The Texture Map object attribute `XGL_TMAP_COORD_SOURCE` specifies the source for the texture coordinates within a primitive's vertex data. For example, if the vertex values are used as the texture coordinates, the point's *x* and *y* values in world coordinates are used for *u* and *v* values of the texture coordinates. If the floating point data is used, the attributes `XGL_TMAP_T0_INDEX` and `XGL_TMAP_T1_INDEX` define the indices into the vertex data array. For information on `XGL_TMAP_COORD_SOURCE`, see "Specifying the Source of Texture Coordinates" on page 471.
- Data Map Texture object – The point type must be one of the `DATA` point types, such as *Xgl_pt_data_f3d*. Data point types include *u* and *v* values that correspond to the coordinate space of the texture image. The attributes `XGL_DMAP_TEXTURE_U_INDEX` and `XGL_DMAP_TEXTURE_V_INDEX` define the indices into the vertex data array of the primitives and indicate which of the data values to use as *u* and *v* texture coordinates.

Using Data Point Types for Texture Mapping

The coordinate space of the texture MIP map that *u* values correspond to is a space that places 0.0 at the left edge of the texture and 1.0 at the right edge of the texture. Similarly for *v*, the coordinate space of the texture MIP map that *v* values use is a space that places 0.0 at the bottom edge of the texture and 1.0 at the top edge of the texture.

If an application were to map a texture completely onto a four-sided polygon in a simple manner with no wrapping but covering the entire polygon, it would specify at the lower left vertex of the polygon the *u,v* pair of (0.0, 0.0), at the upper left vertex (0.0, 1.0), at the upper right vertex (1.0, 1.0), and at the lower right vertex (1.0, 0.0). The following code fragment shows the point data for this mapping.

```
Xgl_pt_list       pl[1];
float             pts[20];

pl[0].pt_type = XGL_PT_DATA_F3D;
pl[0].bbox = NULL;
pl[0].num_pts = 4;
pl[0].num_data_values = 2;
pl[0].pts.data_f3d = (Xgl_pt_data_f3d*) &pts[0];

/*
 * For the first point:
 * {x,y,z,u,v} = {100.0, 100.0, 0.0, 0.0, 0.0}
 */
pts[0] = 100.0;
pts[1] = 100.0;
pts[2] = 0.0;
pts[3] = 0.0;
pts[4] = 0.0;

/*
 * For the second point:
 * {x,y,z,u,v} = {100.0, 300.0, 0.0, 0.0, 1.0}
 */
pts[5] = 100.0;
pts[6] = 300.0;
pts[7] = 0.0;
pts[8] = 0.0;
pts[9] = 1.0;

/*
 * For the third point:
 * {x,y,z,u,v} = {300.0, 300.0, 0.0, 1.0, 1.0}
 */
pts[10] = 300.0;
pts[11] = 300.0;
pts[12] = 0.0;
pts[13] = 1.0;
pts[14] = 1.0;

/*
 * For the fourth point:
 * {x,y,z,u,v} = {300.0, 100.0, 0.0, 1.0, 0.0}
 */
pts[15] = 300.0;
pts[16] = 100.0;
pts[17] = 0.0;
```

```
pts[18] = 1.0;
pts[19] = 0.0;
```

Figure 17-1 shows a polygon with texturing.

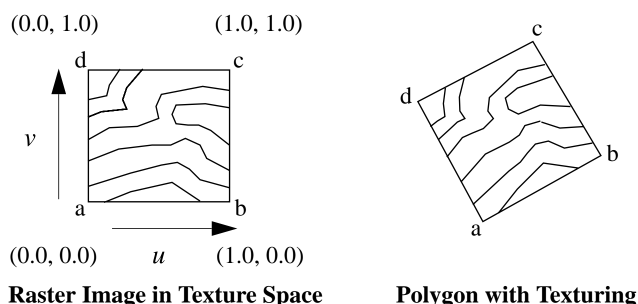


Figure 17-1 Polygon With Texturing

As a second and more involved example, if the application were to map a texture onto a four-sided polygon so that the texture appeared once in the center of the polygon with a border of the original polygon visible around the sides of the texture, then it might specify at the lower left vertex of the polygon the u,v pair of (-0.5, -0.5), at the upper left vertex (-0.5, 1.5), at the upper right vertex (1.5, 1.5), and at the lower right vertex (1.5, -0.5).

If *u* and *v* values are not assigned to polygon vertices in a linear way, that is, if there is no linear transformation that maps *u,v* coordinates to the polygon's vertices, then a textured polygon might look different when rendered on different graphics accelerators. This is because some graphics hardware devices break up more complex polygons into triangles, and if the mapping is non-linear, all the triangles will not be textured the same from device to device.

Note – For any frame buffer, some primitives are accelerated and some may go through software rendering. The inherent differences in the rendering may lead to pixel imperfection when the two types of rendering are intermixed in the same scene.

17

Texture Mapping Example Program

The following program illustrates how to add texture mapping to XGL polygons and how to use the Texture Map object and the MipMap Texture object. The program draws a single polygon without texture, then draws the polygon again with `texture_op` set to `XGL_TEXTURE_OP_BLEND`, and finally redraws the polygon after rotation with the `texture_op` set to `XGL_TEXTURE_OP_MODULATE`. To compile the program, type `make texture`.

Code Example 17-1 Texture Mapping Example

```
/*
 * texture.c
 */

#include <stdio.h>
#include <xgl/xgl.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Xos.h>

static  Display   *display;  /* the connection to the server */
static  int       screen;    /* the screen within the display */
static  Visual    *vis;      /* the visual within the screen */
static  Colormap   cmap;     /* the colormap within the visual */
static  Window     win ;     /* the window */
static  GC         g;        /* the graphics context */
static  Window     create_window(int, int, int, int) ;
static  void       set_properties(Window, int, int, int, int,
                             char*, int, char**);
static u_long      black, white;

main(
    int                             argc,
    char                            *argv[])
{
    Xgl_object                      sys_st;
    Xgl_object                      tmap;
    Xgl_object                      tmap;
    Xgl_object                      ctx;
    Xgl_pt_list                     pl[1];
    float                           pts[20];
    Xgl_object                      mem_ras_front;
    Xgl_object                      front_tm;
```

```
  Xgl_texture_boundary            u_bound, v_bound;
  Xgl_usgn32*                     addr;
  Xgl_texture_general_desc        tm_desc;
  Xgl_render_component_desc*      render_desc;
  Xgl_usgn32                      height = 8;
  Xgl_usgn32                      width = 8;
  Xgl_usgn32                      depth = 32;
  Xgl_usgn32                      i, j;
  Xgl_usgn32*                     tmp_addr;
  Xgl_object                      ras = NULL;
  Xgl_boolean                     tmap_switches[1];
  float                           val = 45.0/180.0 * 3.14;
  char                            hit_return[20];
  Xgl_color                       edge_color;
  Xgl_object                      view_trans;


  Xgl_obj_desc                  obj_desc ;
  Xgl_X_window                  win_desc ;
  Xgl_matrix_f2d                mat;


if ( ( display = XOpenDisplay(NULL) ) == NULL) {
        (void) fprintf(stderr, "display no good! %X\n", display);
        exit(-1);

      screen = DefaultScreen(display);
      vis = XDefaultVisual(display, screen) ;
      cmap = DefaultColormap(display, screen) ;
      win = create_window(200,200,500,500) ;
      set_properties(win,200,200,500,500,"Texture Example",
                    argc,argv) ;

      XMapWindow(display, win);
      XSync(display,False) ;

  win_desc.X_display = display ;
  win_desc.X_screen = screen ;
  win_desc.X_window = win ;
  obj_desc.win_ras.type = XGL_WIN_X ;
  obj_desc.win_ras.desc = (void *)&win_desc ;


  /* Calling xgl_open() */
  sys_st = xgl_open(XGL_SYS_ST_ERROR_DETECTION, 1, 0);
```

```
/* Create a window raster of color type RGB */
ras = xgl_object_create(sys_st, XGL_WIN_RAS, &obj_desc,
                        XGL_DEV_COLOR_TYPE, XGL_COLOR_RGB, 0);

/* Create a 3d context */
ctx = xgl_object_create(sys_st, XGL_3D_CTX, NULL,NULL);

/* Set the Hidden line hidden surface removal to Z Buffer */
xgl_object_set(ctx, XGL_CTX_DEVICE, ras, XGL_3D_CTX_HLHSR_MODE,
               XGL_HLHSR_ZBUFFER, NULL);

xgl_object_set(ctx, XGL_CTX_NEW_FRAME_ACTION,
   XGL_CTX_NEW_FRAME_CLEAR| XGL_CTX_NEW_FRAME_HLHSR_ACTION, 0);
xgl_context_new_frame(ctx);

/* Create a mem ras object; 8x8 32-bits deep */
mem_ras_front = xgl_object_create(sys_st, XGL_MEM_RAS, NULL,
                             XGL_DEV_COLOR_TYPE, XGL_COLOR_RGB,
                              XGL_RAS_DEPTH, depth,
                              XGL_RAS_WIDTH, width,
                              XGL_RAS_HEIGHT, height, NULL);

/* get memory address of the raster */
xgl_object_get(mem_ras_front, XGL_MEM_RAS_MEMORY_ADDRESS,
               &addr);

/* write texture into memory */
tmp_addr = addr;

/*
 * Assume that width is a multiple of
 * 2 for this simple example
 * Write a checked board pattern into memory
 */
for (i = 0; i < height; i += 2) {
   for (j = 0; j < width; j+= 2) {
         *tmp_addr++ =  0xFFFFFFFF;
         *tmp_addr++ = 0x00000000;
    }
    for (j = 0; j < width; j+= 2) {
         *tmp_addr++ =  0x00000000;
         *tmp_addr++ = 0xFFFFFFFF;
    }
}
```

```
/* Create a front MipMap Texture object */;
front_tm = xgl_object_create(sys_st, XGL_MIPMAP_TEXTURE,
                            NULL, 0);
/* Do the filtering operation on the front mipmap */

/* Set the number of levels in front MipMap Texture object */
xgl_object_set(front_tm, XGL_MIPMAP_TEXTURE_LEVELS, 3, NULL);


/* Set the boundary conditions for generating the MIP map */
u_bound = XGL_TEXTURE_BOUNDARY_WRAP;
v_bound = XGL_TEXTURE_BOUNDARY_WRAP;

/*
 * Build  a MIP map pyramid of level 3
 * from the memory raster
 */
xgl_mipmap_texture_build(front_tm, mem_ras_front, u_bound,
                      v_bound);

/*
 * Once the texture is created,
 * mem_ras has no impact on the texture map
 * They can be modified or destroyed
 */
xgl_object_destroy(mem_ras_front);

/* Create the front Texture Map Object
 * For this example, use the attribute defaults:
 * XGL_TMAP_T0_INDEX         : 0
 * XGL_TMAP_T1_INDEX         : 1
 * XGL_TMAP_PARAM_TYPE   : XGL_TEXTURE_PARAM_EXPLICIT
 */
tmap = xgl_object_create(sys_st, XGL_TMAP,NULL, NULL);

/* Fill in the texture descriptor structure */
tm_desc.texture_type = XGL_TEXTURE_TYPE_MIPMAP;

/* Assign the front MipMap Texture Object and  Texture object */
tm_desc.texture_info.mipmap.texture_map = front_tm;

/* Max u and Max v frequency */
tm_desc.texture_info.mipmap.max_u_freq = 1.0;
tm_desc.texture_info.mipmap.max_v_freq = 1.0;
```

```
/* Boundary conditions */
tm_desc.texture_info.mipmap.u_boundary = u_bound;
tm_desc.texture_info.mipmap.v_boundary = v_bound;

/* Assign Boundary color value as Green */
tm_desc.texture_info.mipmap.boundary_values[0] = 0;
tm_desc.texture_info.mipmap.boundary_values[1] = 255;
tm_desc.texture_info.mipmap.boundary_values[2] = 0;
tm_desc.texture_info.mipmap.boundary_values[3] = 0;

/* Set the depth adjustment factor to 0 */
tm_desc.texture_info.mipmap.depth_interp_factor = 0.0;

/* Set the orientation matrix to identity */
tm_desc.texture_info.mipmap.orientation_matrix[0][0] = 1.0;
tm_desc.texture_info.mipmap.orientation_matrix[0][1] = 0.0;
tm_desc.texture_info.mipmap.orientation_matrix[1][0] = 0.0;
tm_desc.texture_info.mipmap.orientation_matrix[1][1] = 1.0;
tm_desc.texture_info.mipmap.orientation_matrix[2][0] = 0.0;
tm_desc.texture_info.mipmap.orientation_matrix[2][1] = 0.0;

/* Sampling methods */
tm_desc.texture_info.mipmap.interp_info.filter1 =
           XGL_TEXTURE_INTERP_MIPMAP_POINT;
tm_desc.txture_info.mipmap.interp_info.filter2 =
           XGL_TEXTURE_INTERP_POINT;

/* Set the color composition information */
tm_desc.comp_info.color_info.num_render_comp_desc = 1;

render_desc =
    &tm_desc.comp_info.color_info.render_component_desc[0];

/*
 * The texture acts on the diffuse color and
 * texture_op is blend
 */
render_desc->comp       = XGL_RENDER_COMP_DIFFUSE_COLOR;
render_desc->texture_op = XGL_TEXTURE_OP_BLEND;

/* Blend colors Red and Blue */
render_desc->op.blend.rgb.c1.r = 1.0;
render_desc->op.blend.rgb.c1.g = 0.0;
render_desc->op.blend.rgb.c1.b = 0.0;

render_desc->op.blend.rgb.c2.r = 0.0;
```

```
render_desc->op.blend.rgb.c2.g = 0.0;
render_desc->op.blend.rgb.c2.b = 1.0;

/* Set this texture desc in the front texture map */
xgl_object_set(tmap, XGL_TMAP_DESCRIPTOR, &tm_desc, 0);

/*
 * Setup a Multi Simple Polygon
 * Pt_type should have DATA
 * Intialize num_data_values
 */
pl[0].pt_type = XGL_PT_DATA_F3D;
pl[0].bbox = NULL;
pl[0].num_pts = 4;
pl[0].num_data_values = 2;
pl[0].pts.data_f3d = (Xgl_pt_data_f3d*)&pts[0];

/*
 * For the first point:
 * {x,y,z,u,v} = {100.0, 100.0, 0.0, 0.0, 0.0}
 */
pts[0] = 100.0;
pts[1] = 100.0;
pts[2] = 0.0;
pts[3] = 0.0;
pts[4] = 0.0;

/*
 * For the second point:
 * {x,y,z,u,v} = {100.0, 300.0, 0.0, 0.0, 1.0}
 */
pts[5] = 100.0;
pts[6] = 300.0;
pts[7] = 0.0;
pts[8] = 0.0;
pts[9] = 1.0;

/*
 * For the third point:
 * {x,y,z,u,v} = {300.0, 300.0, 0.0, 1.0, 1.0}
 */
pts[10] = 300.0;
pts[11] = 300.0;
pts[12] = 0.0;
pts[13] = 1.0;
pts[14] = 1.0;
```

```
/*
 * For the fourth point:
 * {x,y,z,u,v} = {300.0, 100.0, 0.0, 1.0, 0.0}
 */
 pts[15] = 300.0;
 pts[16] = 100.0;
 pts[17] = 0.0;
 pts[18] = 1.0;
 pts[19] = 0.0;

 /* Assign Texture Map Object to the 3D context*/
 xgl_object_set(ctx,
                XGL_3D_CTX_SURF_FRONT_TMAP_NUM, 1,
                XGL_3D_CTX_SURF_FRONT_TMAP, &tmap,
                NULL);

 /*
  * Draw the polygon without texture on it
  * by setting the texture map switch to off
  */
 dmap_texture_switches[0] = FALSE;
 xgl_object_set(ctx, XGL_3D_CTX_SURF_FRONT_TMAP_SWITCHES,
                 tmap_switches, NULL);

 /* Turn the edge on and set the edge color */
 edge_color.rgb.r = 1.0;
 edge_color.rgb.g = 1.0;
 edge_color.rgb.b = 1.0;
 xgl_object_set(ctx, XGL_CTX_EDGE_COLOR, &edge_color,
                XGL_CTX_SURF_EDGE_FLAG, TRUE, NULL);

 /* Render an untextured multi simple polygon */
 xgl_multi_simple_polygon(ctx,
       XGL_FACET_FLAG_SIDES_ARE_4|XGL_FACET_FLAG_SHAPE_CONVEX,
        NULL, NULL,1, pl);

 xgl_context_post(ctx,TRUE);

 fprintf(stderr, "Hit Return to Continue > ");
 fscanf(stdin, "%c", hit_return);

 xgl_context_new_frame(ctx);

/*
 * Draw the polygon with texture on
```

```
* by setting the texture map switch to on
*/
dmap_texture_switches[0] = TRUE;
xgl_object_set(ctx, XGL_3D_CTX_SURF_FRONT_TMAP_SWITCHES,
               tmap_switches, NULL);

/* Render a textured multi simple polygon */
xgl_multi_simple_polygon(ctx,
      XGL_FACET_FLAG_SIDES_ARE_4|XGL_FACET_FLAG_SHAPE_CONVEX,
      NULL, NULL,1, pl);

xgl_context_post(ctx,TRUE);

fprintf(stderr, "Hit Return to Continue > ");
fscanf(stdin, "%c", hit_return);

xgl_context_new_frame(ctx);

/* Rotate the object by setting the view transformation */
xgl_object_get(ctx, XGL_CTX_VIEW_TRANS, &view_trans);
xgl_transform_rotate(view_trans, val, XGL_AXIS_Z,
                      XGL_TRANS_REPLACE);
xgl_transform_rotate(view_trans, val, XGL_AXIS_X,
                      XGL_TRANS_POSTCONCAT);
xgl_transform_rotate(view_trans, val, XGL_AXIS_Y,
                      XGL_TRANS_POSTCONCAT);

/* Change the texture_op to Modulate */
render_desc->texture_op = XGL_TEXTURE_OP_MODULATE;

/* Set this texture desc in the front texture map */
xgl_object_set(tmap, XGL_TMAP_DESCRIPTOR, &tm_desc, 0);

/* Render a textured multi simple polygon */
xgl_multi_simple_polygon(ctx,
        XGL_FACET_FLAG_SIDES_ARE_4|XGL_FACET_FLAG_SHAPE_CONVEX,
        NULL, NULL,1, pl);

xgl_context_post(ctx,TRUE);

/* Wait for the user to hit return */
fprintf(stderr, "Hit Return to Quit > ");
fscanf(stdin, "%c", hit_return);

/* Destroy all created objects */
xgl_object_destroy(tmap);
```

```
    xgl_object_destroy(front_tm);
    xgl_object_destroy(ctx);
    xgl_object_destroy(ras);

    /* call xgl_close */
  xgl_close(sys_st);
}

static Window
create_window(
        int     x,
        int     y,
        int     w,
        int     h)
{
        XSetWindowAttributes    attributes ;

        black = BlackPixel(display, screen) ;
        white = WhitePixel(display, screen) ;

        attributes.background_pixel = black ;
        attributes.border_pixel = black ;
        attributes.colormap = cmap ;

        return XCreateWindow(display, DefaultRootWindow(display),
                x, y, w, h, 5,
                CopyFromParent, CopyFromParent, vis,
               CWBackPixel|CWBorderPixel|CWColormap, &attributes) ;
}

static void
set_properties(
        Window  win,
        int     x,
        int     y,
        int     w,
        int     h,
        char    *label,
        int     argc,
        char    **argv)
{
        XSizeHints      size_hints;
        XWMHints        hints ;

        size_hints.x = x-5 ;
        size_hints.y = y-26 ;
```

```
        size_hints.width = w ;
        size_hints.height = h ;
        size_hints.flags = PPosition|PSize ;

        XSetStandardProperties(display, win, label, label,
                NULL, argv, argc, &size_hints );

        XSelectInput(display, win, ExposureMask | KeyPressMask |
                ButtonPressMask | StructureNotifyMask );

        hints.flags = InputHint ;
        hints.input = True ;
        XSetWMHints(display,win, &hints) ;
}
```

Changes to the XGL 3.2 Library

This appendix provides information on the differences between the XGL 3.1 product and the current XGL product. It lists and briefly describes additions, changes, and deletions to the XGL library. For complete information on all XGL operators, attributes, and data structures, see the *XGL Reference Manual*.

New Operators

Table A-1 New Operators

Attribute	Description
`xgl_gcache_triangle_list()`	New Gcache operator that renders a triangle list into a specified Gcache object.

Changed Attributes

Table A-2 Changed Attributes

Attribute	Description
XGL_CTX_RASTER_FILL_STYLE	Added the following value to provide stencil text: XGL_RAS_FILL_STENCIL
XGL_3D_CTX_SURF_TMAP_PERSP_CORRECTION	Added the value XGL_TEXTURE_PERSP_NO_INTERP. This value provides vertex-level texture mapping.
XGL_TMAP_PARAM_TYPE	Added the following values that provide environment texture mapping: XGL_TEXTURE_PARAM_REFLECT_SPHERE_EC, XGL_TEXTURE_PARAM_REFLECT_SPHERE_WC, XGL_TEXTURE_PARAM_LINEAR_EC

Changed Data Structures

Table A-3 Changed Data Structures

Data Structure	Description
Xgl_raster_fill_style	XGL_RAS_FILL_STENCIL
Xgl_texture_param_type	XGL_TEXTURE_PARAM_REFLECT_SPHERE_EC XGL_TEXTURE_PARAM_RELFECT_SPHERE_WC XGL_TEXTURE_PARAM_LINEAR_EC
Xgl_texture_persp_correction	XGL_TEXTURE_PERSP_NO_INTERP

Changes to the XGL Library From XGL 3.0 through XGL 3.1

This appendix provides information on changes to the XGL library from XGL 3.0 through XGL 3.1. There are two sections:

- Changes to the XGL library at XGL 3.1
- Changes to the XGL library from XGL 3.0 through XGL 3.0.2

The appendix lists and briefly describes additions, changes, and deletions to the XGL library. Operators, attributes, and data structures that are not listed in this appendix remained unchanged. For complete information on all XGL operators, attributes, and data structures, see the *XGL Reference Manual*.

Changes to the XGL Library at XGL 3.1

This section provides information on the differences between the XGL 3.0.2 product and the XGL 3.1 product.

Changed Operators

Table B-1 Changed Operator

Operator	Description
`xgl_inquire()`	Added areturn value to indicate the X extensions supported by the frame buffer. Currently, only a query to the multibuffer extension to X (MBX) is provided.

Deleted Operators

Table B-2 Changed Operator

Operator	Description
xgl_context_copy_raster()	This operator has been replaced by xgl_context_copy_buffer().

New Attributes

Table B-3 New Attributes

Attribute	Description
XGL_3D_CTX_SURF_FRONT_TMAP_NUM XGL_3D_CTX_SURF_BACK_TMAP_NUM XGL_3D_CTX_SURF_FRONT_TMAP XGL_3D_CTX_SURF_BACK_TMAP XGL_3D_CTX_SURF_FRONT_TMAP_SWITCHES XGL_3D_CTX_SURF_BACK_TMAP_SWITCHES	Context attributes that apply one or more textures to the front or back of geometric objects. These attributes specify a list of Texture Map objects, specify the number of Texture Map objects, and specify which of the Texture Map objects should be applied.
XGL_3D_SURF_TMAP_PERSP_CORRECTION	Specifies the method used to compute texture coordinate values for surface interiors.
XGL_3D_CTX_SURF_LIGHTING_NORMAL_FLIP	Defines how surface normals are treated for lighting.
XGL_PCACHE_CONTEXT	Associates a Pcache object with a 2D or 3D Context.
XGL_CTX_PAINT_TYPE	Sets the paint type on a transparent overlay window.
XGL_CTX_NEW_FRAME_PAINT_TYPE	Defines the paint type used when an overlay windows is cleared with xgl_context_new_frame().
XGL_TMAP_DESCRIPTOR	Associates the texture descriptor structure with the Texture Map object.
XGL_TMAP_COORD_SOURCE	Specifies the initial source within a primitive's vertex data of the texture coordinate.
XGL_TMAP_DOMAIN	Specifies the domain to which the texture is to be applied.

Table B-3 New Attributes

Attribute	Description
XGL_TMAP_PARAM_TYPE	Defines the parameterization method for mapping a texture to a 3D surface.
XGL_TMAP_T0_INDEX XGL_TMAP_T1_INDEX	When the source of the texture coordinates is the floating point data, these attributes define the indices into the vertex data array of the surface primitives.
XGL_WIN_RAS_MULTIBUFFER	Requests multibuffering using the MBX extension to X for a Window Raster.
XGL_WIN_RAS_MBUF_DRAW	Specifies the buffer to read and write pixels into when using MBX multibuffering.

Changed Attributes

Table B-4 Changed Attributes

Attribute	Description
XGL_CTX_RENDER_BUFFER	The value XGL_RENDER_NONE was removed.

Deleted Attributes

Table B-5 Deleted Attributes

Attribute	Description
XGL_CMAP_COLOR_CUBE_RSIZE XGL_CMAP_COLOR_CUBE_GSIZE XGL_CMAP_COLOR_CUBE_BSIZE	Deleted.Use the XGL_CMAP_COLOR_CUBE_SIZE attribute, which contains all three color cube size values.
XGL_CTX_RENDERING	Deleted. Use XGL_CTX_RENDER_BUFFER.
XGL_CTX_VIEW_MODEL_DATA_TYPE	Deleted.
XGL_DEV_PICK_BUFFER_SIZE	Deleted.
XGL_LPAT_COORD_SYS	Deleted.
XGL_SYS_ST_ERROR_FILE_NAME	Deleted.
XGL_3D_CTX_SURF_FRONT_LIGHT_TYPE XGL_3D_CTX_SURF_BACK_LIGHT_TYPE	Deleted.

New Data Structures

Table B-6 Deleted Attributes

Data Structure	Description
Xgl_paint_type	Added new paint type structure for transparent overlay window support.
Xgl_texture_persp_correction	Added enumerated type to support texture map perspective correction.
Xgl_texture_coord_source	Added enumerated type to define the source of the texture coordinates in the primitive's point data.
Xgl_texture_domain	Added enumerated type to define the texture domain.
Xgl_texture_type	Added enumerated type to define the type of texture mapping.

Table B-6 Deleted Attributes

Data Structure	Description
Xgl_texture_general_mipmap_desc	Added new MipMap descriptor structure for the Texture Map object.
Xgl_texture_comp_info	Added data structure to specify how the texture map is blended with the primitive's data.
Xgl_texture_general_desc	Added new texture descriptor structure for the Texture Map object.

Changed Data Structures

Table B-7 Deleted Attributes

Data Structure	Description
Xgl_ctx_new_frame_action	The value XGL_CTX_NEW_FRAME_POINT_TYPE_ACTION has been added to Xgl_ctx_new_frame_action.
Xgl_render_mode	The value XGL_RENDER_NONE has been removed from Xgl_render_mode.
Xgl_texture_boundary	Added new values to support texture mapping.
Xgl_texture_interp_method	Added value to support texture mapping.
Xgl_texture_op	Added new values to support texture mapping.
Xgl_texture_color_comp_info	Added channel information to the color composition method.

Changes From XGL 3.0 Through XGL 3.0.2

This section provides information on the changes in the XGL library from XGL 3.0 to XGL 3.0.2. General information on updating an XGL application from XGL 2.0 to XGL 3.x is provided, and specific additions, changes, and deletions to the library are listed and described. Application program developers can use this information to port XGL 2.0 applications to the XGL 3.x library.

Using the xgl_changes Script

A script is available to assist application developers in porting applications from XGL 2.0 to XGL 3.x. Developers can use this script to flag source code changes that need to be made to run applications with XGL 3.x. The script scans a source code file (`.c` or `.h`) for changes. All attributes, functions, operators, and structures that changed from XGL 2.0 to XGL 3.0.2 or that were deleted are flagged, and the line number of the source file is given.

The script is composed of ten script files. The primary script, `xgl_changes`, calls files named `xgl_changes##`, where `##` is `00-08`. These files are part of the `SUNWxglh` package and are installed in `$XGLHOME/demo`, if `$XGLHOME` is specified, or in the default area `/opt/SUNWits/Graphics-sw/xgl/demo`, if `$XGLHOME` is not specified. To use the script, key in the following:

```
xgl_changes <single_file> | more
```

Programming Notes for XGL 3.x

This section provides tips on porting applications to the 3.x versions of the XGL library.

- The application must call `xgl_open()` before calling the operator `xgl_inquire()`, or an error will result.
- The application must set `XGL_CMAP_COLOR_TABLE_SIZE` prior to setting `XGL_CMAP_COLOR_TABLE`. The value of `XGL_CMAP_COLOR_TABLE_SIZE` should be greater than or equal to the sum of the starting index and the length structure elements passed to XGL with the `XGL_CMAP_COLOR_TABLE` attribute.
- `XGL_DEV_COLOR_TYPE`, if set during device creation time, must be the first attribute in the attribute list.

- It is important to follow the exact order of functions called and attributes set when setting a color map with the attribute XGL_CMAP_NAME. Please refer to the man page for XGL_CMAP_NAME.
- To convert XGL colors to pixel values, use a color mapper (XGL_CMAP_COLOR_MAPPER) rather than an inverse color mapper (XGL_CMAP_INVERSE_COLOR_MAPPER) as in XGL 2.0. The inverse color mapper converts pixel values to XGL colors.
- The file extension for the XGL 3.x fonts in the stroke font directory is *.font. The XGL font files were changed at the XGL 3.0 release to improve inter-character spacing and increase text readability. Links are provided from the XGL 2.0 font file names (file name suffix *.phont) to the XGL 3.x font file names to accommodate programs written for the XGL 2.0 library.
- The System State object type is now *Xgl_sys_state*. For XGL 2.0, the System State object type was *Xgl_sys_st*.
- Some attributes have changed data type. For certain attributes, if the application passes a pointer to a float structure, XGL 3.x will be expecting a double. Be sure that you read the sections that follow and update your application appropriately.
- CGM Device functionality is no longer directly supported in XGL. CGM functionality may be indirectly provided in a loadable device pipeline through the XGL 3.0.2 XGL_STREAM device functionality. The Stream Device offers new extension possibilities, including CGM availability through third party software developers.

Note – The file xgl_cgm-2.0.h is provided to allow backward source compatibility for applications that were using XGL 2.0 CGM functionality. This file will allow applications using XGL 2.0 CGM functionality to compile and link, but it will not provide CGM output.

New Operators

Table B-8 New Operators

Operator	Description
xgl_context_accumulate()	Accumulates pixel information to generate antialiased images.
xgl_context_clear_accumulation()	Clears an accumulation buffer.
xgl_context_check_bbox()	Calculates the geometry status of a bounding box.
xgl_context_flush()	Causes pending graphics primitives to be posted or asynchronous processing to conclude. Flushes the XGL system's use of an application's data and ensures synchronization of XGL and device processing. This replaces and extends xgl_context_post().
xgl_mipmap_texture_build()	Creates a mipmap pyramid of memory rasters. The mipmap is used as the texture in texturing operations.
xgl_nurbs_surface()	Renders a NURBS surface. The NURBS surface can be trimmed, and isoparametric lines can be displayed on the surface.
xgl_transform_write_specific()	Describes a transformation matrix that an application is passing to XGL. Specifying the matrix type can significantly improve the performance of operations involving transformations.
xgl_triangle_list()	Renders a 3D triangle list. Triangle lists can consist of connected triangles arranged as a triangle strip, connected triangles arranged as a triangle star, or unconnected triangles.

Changed Operators

Table B-9 Changed Operators

Operator	Description
`xgl_context_copy_buffer()`	Copies a rectangular block of pixels from a buffer in the source Raster to one or more of the Raster buffers associated with the destination Context. The source buffer is specified by the attribute `XGL_RAS_SOURCE_BUFFER`. The destination buffer is specified by the Context attribute `XGL_CTX_RENDER_BUFFER`. Copying is only supported between buffers of the same type. This replaces and extends the `xgl_copy_raster()` operator.
`xgl_context_copy_raster()`	This operator has been superseded by `xgl_context_copy_buffer()`, but it is retained for backward compatibility.
`xgl_context_pop()`	Changed to allow depth-cue attributes to be pushable attributes.
`xgl_context_post()`	This operator has been superseded by `xgl_context_flush()`, but it is retained by backward compatibility.
`xgl_context_push()`	Changed to allow depth-cue attributes to be pushable attributes.
`xgl_context_set_multi_pixel()`	The value of the `count` argument has changed from a signed 32-bit integer to an unsigned 32-bit integer.
`xgl_context_set_pixel_row()`	The values of the `startcolumn`, `row`, and `count` arguments have changed from signed 32-bit integers to unsigned 32-bit integers.
`xgl_gcache_nu_bspline_curve()`	This operator has been superseded by `xgl_gcache_nurbs_curve()`, but it is retained for backward compatibility.
`xgl_gcache_nurbs_curve()`	This operator replaces the XGL 2.0 `xgl_gcache_nu_bspline_curve()` operator. However, the `xgl_gcache_nu_bspline_curve()` operator is retained for backward compatibility.

Table B-9 Changed Operators *(Continued)*

Operator	Description
`xgl_inquire()`	Added an *Xgl_sys_state* argument.
`xgl_nu_bspline_curve()`	This operator has been superseded by `xgl_nurbs_curve()`, but it is retained for backward compatibility.
`xgl_nurbs_curve()`	This operator replaces the XGL 2.0 `xgl_nu_bspline_curve()` operator. However, the `xgl_nu_bspline_curve()` operator is retained for backward compatibility.
`xgl_object_create()`	Added a *type* parameter `XGL_STREAM` used in the creation of the Stream Device object.
`xgl_object_create()`	Added *type* parameters `XGL_DMAP_TEXTURE` and `XGL_MIPMAP_TEXTURE` used for texture mapping.
`xgl_transform_read()`	Accepts matrices with a data type of double if `XGL_TRANS_DATA_TYPE` is `XGL_DATA_DBL`.
`xgl_transform_rotate()`	The value of the `angle` argument has changed from `float` to `double`.
`xgl_transform_scale()`	Changed to accept double point types for the *Xgl_pt* argument.
`xgl_transform_translate()`	Changed to accept double point types for the *Xgl_pt* argument.
`xgl_transform_write()`	This operator has been superseded by `xgl_transform_write_specific()` but retained for backward compatibility.

Deleted Operators

Table B-10 Deleted Operators

Operator	Description
xgl_2d_context_create()	Use xgl_object_create().
xgl_3d_context_create()	Use xgl_object_create().
xgl_color_map_create()	Use xgl_object_create().
xgl_light_create()	Use xgl_object_create().
xgl_line_pattern_create()	Use xgl_object_create().
xgl_marker_create()	Use xgl_object_create().
xgl_memory_raster_device_create()	Use xgl_object_create().
xgl_stroke_font_create()	Use xgl_object_create().
xgl_transform_create()	Use xgl_object_create().
xgl_window_raster_device_create()	Use xgl_object_create().

New Attributes

Table B-11 New Attributes

Attribute	Description
XGL_3D_CTX_ACCUM_OP_DEST	Context attributes that specifies the buffer to be used as the destination buffer during an accumulation operation.
XGL_3D_CTX_JITTER_OFFSET	Context attribute that specifies the amount by which geometry should be offset in device coordinates before drawing. Used when rendering an image before accumulating into an accumulation buffer.
XGL_3D_CTX_Z_BUFFER_COMP_METHOD	Context attribute that specifies the comparison technique used when the Z-buffer is updated.
XGL_3D_CTX_Z_BUFFER_WRITE_MASK	Context attribute that defines which bit planes of the Z-buffer are written.
XGL_3D_CTX_SURF_TRANSP_METHOD XGL_3D_CTX_SURF_FRONT_TRANSP XGL_3D_CTX_SURF_BACK_TRANSP XGL_3D_CTX_BLEND_DRAW_MODE XGL_3D_CTX_BLEND_FREEZE_Z_BUFFER XGL_3D_CTX_SURF_TRANSP_BLEND_EQ	Context attributes that enable applications to render transparent surfaces. The attributes define the transparency method and, if appropriate, the blending equations.
XGL_3D_CTX_SURF_FRONT_DMAP_NUM XGL_3D_CTX_SURF_BACK_DMAP_NUM XGL_3D_CTX_SURF_FRONT_DMAP XGL_3D_CTX_SURF_BACK_CMAP XGL_3D_CTX_SURF_FRONT_DMAP_SWITCHES XGL_3D_CTX_SURF_BACK_DMAP_SWITCHES	Context attributes that apply one or more textures to the front or back of geometric objects. These attributes specify a list of Data Map Texture objects, specify the number of Data Map Texture objects, and specify which of the Data Map Texture objects should be applied.
XGL_CTX_GEOM_DATA_IS_VOLATILE	Context attribute that indicates that the application program guarantees that the geometry data will not be modified or destroyed until a subsequent `xgl_context_flush()` call.
XGL_CTX_EDGE_AA_BLEND_EQ XGL_CTX_EDGE_AA_FILTER_WIDTH XGL_CTX_EDGE_AA_FILTER_SHAPE	Context attributes that control whether surface edges are antialiased and define the method of antialiasing.
XGL_CTX_LINE_AA_BLEND_EQ XGL_CTX_LINE_AA_FILTER_WIDTH XGL_CTX_LINE_AA_FILTER_SHAPE	Context attributes that control whether lines are antialiased and define the method of antialiasing.

Table B-11 New Attributes *(Continued)*

Attribute	Description
XGL_CTX_MARKER_AA_BLEND_EQ XGL_CTX_MARKER_AA_FILTER_WIDTH XGL_CTX_MARKER_AA_FILTER_SHAPE	Context attributes that control whether markers are antialiased and define the method of antialiasing.
XGL_CTX_MC_TO_DC_TRANS	Read-only Transform attribute that transforms points from Model Coordinates to Device Coordinates.
XGL_CTX_NURBS_CURVE_APPROX XGL_CTX_NURBS_CURVE_APPROX_VAL	Context attributes that control the tessellation of a NURBS curve.
XGL_CTX_NURBS_SURF_APPROX XGL_CTX_NURBS_SURF_APPROX_VAL_{U,V} XGL_CTX_NURBS_SURF_ISO_CURVE_ PLACEMENT XGL_CTX_NURBS_SURF_ISO_CURVE_{U,V}_NUM XGL_CTX_NURBS_SURF_PARAM_STYLE	Context attributes that control the tessellation of a NURBS surface and that define how isoparametric curves are drawn on a NURBS surface.
XGL_CTX_RENDER_BUFFER	Context attribute that controls whether primitives are rendered on the Device associated with the Context.
XGL_CTX_RENDERING_ORDER	Context attribute that allows parallel graphics devices to render in any order, as opposed to rendering in the application's given order.
XGL_CTX_STEXT_AA_BLEND_EQ XGL_CTX_STEXT_AA_FILTER_WIDTH XGL_CTX_STEXT_AA_FILTER_SHAPE	Context attributes that control whether stroke text is antialiased and define the method of antialiasing.
XGL_CTX_SURF_AA_BLEND_EQ XGL_CTX_SURF_AA_FILTER_WIDTH XGL_CTX_SURF_AA_FILTER_SHAPE	Context attributes that control whether surfaces are antialiased and define the method of antialiasing.
XGL_DEV_COLOR_MAP	A read-only Device attribute that returns the actual workstation color type.
XGL_DMAP_TEXTURE_DESCRIPTORS XGL_DMAP_TEXTURE_NUM_DESCRIPTORS XGL_DMAP_TEXTURE_ORIENTATION_MATRIX XGL_DMAP_TEXTURE_PARAM_TYPE XGL_DMAP_TEXTURE_U_INDEX XGL_DMAP_TEXTURE_V_INDEX	Data Map Texture object attributes that describe how a texture is applied to a primitive.

Table B-11 New Attributes *(Continued)*

Attribute	Description
XGL_MEM_RAS_Z_BUFFER_ADDR	Memory Raster attribute that returns the starting address of the block of memory for the Z-buffer of a Memory Raster, allowing the application programmer to read and write Z-buffer values.
XGL_MIPMAP_TEXTURE_WIDTH XGL_MIPMAP_TEXTURE_HEIGHT XGL_MIPMAP_TEXTURE_DEPTH XGL_MIPMAP_TEXTURE_LEVELS XGL_MIPMAP_TEXTURE_MEM_RAS_LIST	Mipmap Texture object attributes that define the texture used in texturing operations.
XGL_SYS_ST_ERROR_INFO	A read-only System State attribute that contains state information about the last error that was detected.
XGL_WIN_RAS_BACKING_STORE	Window Raster attribute that allows applications to enable backing store.

Changed Attributes

Table B-12 Changed Attributes

Attribute	Description
XGL_CMAP_COLOR_CUBE_RSIZE XGL_CMAP_COLOR_CUBE_GSIZE XGL_CMAP_COLOR_CUBE_BSIZE	Superseded by the XGL_CMAP_COLOR_CUBE_SIZE attribute, which contains all three color cube size values.
XGL_CMAP_COLOR_MAPPER XGL_CMAP_INVERSE_COLOR_MAPPER	Added a parameter to the function call to the application color mapper function. The new parameter is a Boolean that indicates whether the color mapping should include the pixel mapping array.
XGL_3D_CTX_DEPTH_CUE_REF_PLANES	Changed value type from float [2] to double [2].
XGL_3D_CTX_DEPTH_CUE_COLOR XGL_3D_CTX_DEPTH_CUE_INTERP XGL_3D_CTX_DEPTH_CUE_MODE XGL_3D_CTX_DEPTH_CUE_REF_PLANES XGL_3D_CTX_DEPTH_CUE_SCALE_FACTORS	Changed depth-cue attributes from environment (not pushable) to graphics (pushable) context attributes.
XGL_3D_CTX_SURF_GEOM_NORMAL	Changed quadrilateral mesh geometric normal calculations so that they match the PHIGS PLUS standard.
XGL_CTX_DC_VIEWPORT	Changed value type from *Xgl_bounds_f2d* to *Xgl_bounds_d2d* for 2D, and from *Xgl_bounds_f3d* to *Xgl_bounds_d3d* for 3D.
XGL_CTX_EDGE_WIDTH_SCALE_FACTOR	Default value changed from 1.0 to 0.0.
XGL_CTX_LINE_WIDTH_SCALE_FACTOR	Default value changed from 1.0 to 0.0.
XGL_CTX_MAX_TESSELLATION	Default value changed from 128 to 64.
XGL_CTX_MIN_TESSELLATION	Default value changed from 8 to 1.
XGL_CTX_NURBS_CURVE_APPROX (see Table B-13 for previous name)	Default value changed from XGL_CURVE_CONST_PARAM_SUBDIV to XGL_CURVE_CONST_PARAM_SUBDIV_BETWEEN_ KNOTS.
XGL_CTX_NURBS_SURF_APPROX (see Table B-13 for previous name)	Default value changed from XGL_CURVE_CONST_PARAM_SUBDIV to XGL_CURVE_CONST_PARAM_SUBDIV_BETWEEN_ KNOTS.

Table B-12 Changed Attributes *(Continued)*

Attribute	Description
XGL_CTX_PICK_APERTURE	Changed value type from *Xgl_bounds_f2d* to *Xgl_bounds_d2d* for 2D, and from *Xgl_bounds_f3d* to *Xgl_bounds_d3d* for 3D.
XGL_CTX_SURF_INTERIOR_RULE	The nonzero winding number rule is no longer supported for determining the interior of a polygon. The XGL_NONZERO_WINDING_RULE (nonzero) value of the XGL_CTX_SURF_INTERIOR_RULE attribute can still be set, but the user will get the XGL_EVEN_ODD (even-odd) value instead.
XGL_CTX_RENDERING	This attribute has been superseded by XGL_CTX_RENDER_BUFFER, but it is retained for backward compatibility.
XGL_CTX_RENDER_BUFFER	The value XGL_RENDER_NONE has been added for backward compatibility.
XGL_CTX_VDC_WINDOW	Changed value type from *Xgl_bounds_f2d* to *Xgl_bounds_d2d* for 2D, and from *Xgl_bounds_f3d* to *Xgl_bounds_d3d* for 3D.
XGL_CTX_VIEW_CLIP_BOUNDS	Changed value type from *Xgl_bounds_f2d* to *Xgl_bounds_d2d* for 2D, and from *Xgl_bounds_f3d* to *Xgl_bounds_d3d* for 3D.
XGL_CTX_VIEW_MODEL_DATA_TYPE	Setting this attribute no longer has any affect on the internal transforms on a Context; however, this attribute is retained for backward compatibility.
XGL_DEV_MAXIMUM_COORDINATES	Changed value type from *Xgl_pt_f3d* to *Xgl_pt_d3d* to support double precision.
XGL_DEV_PICK_BUFFER_SIZE	Retained for backward compatibility. Return value is always 0.
XGL_LIGHT_POSITION	Changed value type from *Xgl_pt_f3d* to *Xgl_pt_d3d* to support double precision.

Table B-12 Changed Attributes *(Continued)*

Attribute	Description
XGL_LPAT_COORD_SYS	Line pattern objects can only be defined in the device coordinate system. This attribute is retained for backward compatibility only.
XGL_TRANS_DATA_TYPE	Can be set to a data type of XGL_DATA_DBL for both 2D and 3D.
XGL_SYS_ST_ERROR_NOTIFICATION_FUNCTION	Changed argument list to take only an *Xgl_sys_state* argument.

Renamed Attributes

Table B-13 lists attributes that have been renamed. Note that the old names will continue to work during the lifetime of XGL 3.x; however, after that they will no longer be supported.

Table B-13 Renamed Attributes

Old Name (XGL 2.0)	New Name (XGL 3.x)
XGL_3D_CTX_SURF_APPROX	XGL_CTX_NURBS_SURF_APPROX
XGL_CTX_CURVE_APPROX	XGL_CTX_NURBS_CURVE_APPROX
XGL_CTX_CURVE_APPROX_VALUE	XGL_CTX_NURBS_CURVE_APPROX_VAL
XGL_CTX_MARKER_DESCRIPTION	XGL_CTX_MARKER
XGL_MEM_RAS_MEMORY_ADDRESS	XGL_MEM_RAS_IMAGE_BUFFER_ADDR
XGL_CTX_SFONT_CHARSET_0	XGL_CTX_SFONT_0
XGL_CTX_SFONT_CHARSET_1	XGL_CTX_SFONT_1
XGL_CTX_SFONT_CHARSET_2	XGL_CTX_SFONT_2
XGL_CTX_SFONT_CHARSET_3	XGL_CTX_SFONT_3
XGL_CTX_SFONT_CHAR_ENCODING	XGL_CTX_STEXT_CHAR_ENCODING
XGL_CTX_SFONT_CHAR_EXPANSION_FACTOR	XGL_CTX_STEXT_CHAR_EXPANSION_FACTOR
XGL_CTX_SFONT_CHAR_HEIGHT	XGL_CTX_STEXT_CHAR_HEIGHT
XGL_CTX_SFONT_CHAR_SLANT_ANGLE	XGL_CTX_STEXT_CHAR_SLANT_ANGLE

Table B-13 Renamed Attributes *(Continued)*

Old Name (XGL 2.0)	New Name (XGL 3.x)
XGL_CTX_SFONT_CHAR_SPACING	XGL_CTX_STEXT_CHAR_SPACING
XGL_CTX_SFONT_CHAR_UP_VECTOR	XGL_CTX_STEXT_CHAR_UP_VECTOR
XGL_CTX_SFONT_TEXT_COLOR	XGL_CTX_STEXT_COLOR
XGL_CTX_SFONT_TEXT_PATH	XGL_CTX_STEXT_PATH
XGL_CTX_SFONT_TEXT_PREC	XGL_CTX_STEXT_PRECISION
XGL_CTX_SFONT_TEXT_ALIGN_HORIZ	XGL_CTX_STEXT_ALIGN_HORIZ
XGL_CTX_SFONT_TEXT_ALIGN_VERT	XGL_CTX_STEXT_ALIGN_VERT
XGL_CTX_ANNOT_CHAR_HEIGHT	XGL_CTX_ATEXT_CHAR_HEIGHT
XGL_CTX_ANNOT_CHAR_SLANT_ANGLE	XGL_CTX_ATEXT_CHAR_SLANT_ANGLE
XGL_CTX_ANNOT_CHAR_UP_VECTOR	XGL_CTX_ATEXT_CHAR_UP_VECTOR
XGL_CTX_ANNOT_STYLE	XGL_CTX_ATEXT_STYLE
XGL_CTX_ANNOT_TEXT_ALIGN_HORIZ	XGL_CTX_ATEXT_ALIGN_HORIZ
XGL_CTX_ANNOT_TEXT_ALIGN_VERT	XGL_CTX_ATEXT_ALIGN_VERT
XGL_CTX_ANNOT_TEXT_PATH	XGL_CTX_ATEXT_PATH

Deleted Attributes

Table B-14 Deleted Attributes

Attribute	Description
XGL_CGM_COLOR_MAP	Replaced with XGL_DEV_COLOR_MAP.
XGL_CGM_DEV	Deleted.
XGL_CGM_DESCRIPTION	Deleted.
XGL_CGM_ENCODING	Deleted.
XGL_CGM_PICTURE_DESCRIPTION	Deleted.
XGL_CGM_SCALE_FACTOR	Deleted.
XGL_CGM_SCALE_MODE	Deleted.
XGL_CGM_TYPE	Deleted.
XGL_CGM_VDC_EXTENT	Deleted.
XGL_EDGE_ANTI_ALIASING	Replaced with the new antialiasing edge attributes. See Table B-11.
XGL_LINE_ANTI_ALIASING	Replaced with the new antialiasing line attributes. See Table B-11.
XGL_MARKER_ANTI_ALIASING	Replaced with the new antialiasing marker attributes. See Table B-11.
XGL_RAS_COLOR_MAP	Replaced with XGL_DEV_COLOR_MAP.
XGL_SFONT_ANTI_ALIASING	Replaced with the new antialiasing stroke text attributes. See Table B-11.
XGL_SURF_ANTI_ALIASING	Replaced with the new antialiasing surface attributes. See Table B-11.
XGL_SYS_ST_ERROR_FILE_NAME	Deleted.
XGL_3D_CTX_SURF_FRONT_LIGHT_TYPE	Deleted.
XGL_3D_CTX_SURF_BACK_LIGHT_TYPE	Deleted.

New Data Structures

Table B-15 New Data Structures

Data Structure	Description
Xgl_accum_depth	Added an enumerated type for the accumulation buffer depth values.
Xgl_arc_d3d *Xgl_arc_ad3d*	Added new arc structures to support double precision.
Xgl_bbox_i2d *Xgl_bbox_f2d* *Xgl_bbox_f3d* *Xgl_bbox_d2d* *Xgl_bbox_d3d* *Xgl_bbox_status*	Added six new bounding box data structures. These structures enable the application to tailor memory usage to its specific requirements. For detailed information, see the man pages for the primitives that use bounding boxes, such as `xgl_polygon()`, and `xgl_context_check_bbox()`.
Xgl_blend_eq *Xgl_blend_draw_mode* *Xgl_filter_shape*	Added new structures to support anti-aliasing and transparency.
Xgl_bounds_d1d *Xgl_bounds_d2d* *Xgl_bounds_d3d*	Added new bounds structures to support double precision.
Xgl_circle_d3d *Xgl_circle_ad3d*	Added new circle structures to support double precision.
Xgl_ell_d3d *Xgl_ell_ad3d*	Added new ellipse structures to support double precision.
Xgl_error_category	Added a new set of error categories.
Xgl_error_info	Added a data structure to describe the most recent error.

Table B-15 New Data Structures *(Continued)*

Data Structure	Description
Xgl_pt_d2d *Xgl_pt_color_d2d* *Xgl_pt_flag_d2d* *Xgl_pt_d2h* *Xgl_pt_d3d* *Xgl_pt_color_d3d* *Xgl_pt_normal_d3d* *Xgl_pt_color_normal_d3d* *Xgl_pt_flag_d3d* *Xgl_pt_color_flag_d3d* *Xgl_pt_normal_flag_d3d* *Xgt_pt_color_normal_flag_d3d* *Xgl_pt_d3h*	Added new point structures to support double precision.
Xgl_pt_data_f3d *Xgl_pt_color_data_f3d* *Xgl_pt_normal_data_f3d* *Xgl_pt_color_normal_data_f3d* *Xgl_pt_flag_data_f3d* *Xgl_pt_color_flag_data_f3d* *Xgl_pt_normal_flag_data_f3d* *Xgl_pt_color_normal_flag_data_f3d*	Added new point structures to support texture mapping.
Xgl_nurbs_curve	Added new NURBS curve structure.
Xgl_nurbs_surf *Xgl_nurbs_surf_type* *Xgl_nurbs_surf_geom_desc* *Xgl_nurbs_surf_simple_geom* *Xgl_surf_color_spline*	Added new NURBS surface data structures.
Xgl_rect_d3d *Xgl_rect_ad3d*	Added new rectangle structures to support double precision.
Xgl_surf_approx	Added a surface approximation data type. See the `XGL_CTX_NURBS_SURF_APPROX` man page for more information.

Table B-15 New Data Structures *(Continued)*

Data Structure	Description
Xgl_render_component *Xgl_texture_boundary* *Xgl_texture_interp_method* *Xgl_texture_op* *Xgl_texture_param_type* *Xgl_texture_type* *Xgl_texture_blend_op* *Xgl_render_component_desc* *Xgl_texture_interp_info* *Xgl_texture_color_comp_info* *Xgl_texture_mipmap_desc* *Xgl_texture_desc*	Added data structures for texture mapping.
Xgl_transp_method	Added data structure to support transparency.
Xgl_trim_curve *Xgl_trim_loop* *Xgl_trim_loop_list* *Xgl_trim_curve_approx*	Added data structures for NURBS surface trimming loops.

Changed Data Structures

Table B-16 lists changed enumerated types and data structures. See the man pages for *xgl_enum_type* and *xgl_struct* for more information.

Table B-16 Changed Data Structures

Data Structure	Change
Xgl_annot_style	Has been renamed to `Xgl_atext_style`. The values for this ennumerated type have changed to `XGL_ATEXT_STYLE_NORMAL` and `XGL_ATEXT_STYLE_LINE`.
Xgl_bbox	Removed the *Xgl_bounds_i3d* and *Xgl_bounds_b2d* members from the union of bounding box types in the *Xgl_bbox* structure.
Xgl_curve_approx	Renamed NURBS curve value `XGL_CURVE_CONST_PARAM_SUBDIV` to `XGL_CURVE_UNUSED`.

Xgl_curve_approx	Added new approximation values to *Xgl_curve_approx*. The new approximation values are:
	`XGL_CURVE_METRIC_WC`
	`XGL_CURVE_METRIC_VDC`
	`XGL_CURVE_METRIC_DC`
	`XGL_CURVE_CHORDAL_DEVIATION_WC`
	`XGL_CURVE_CHORDAL_DEVIATION_VDC`
	`XGL_CURVE_CHORDAL_DEVIATION_DC`
	`XGL_CURVE_RELATIVE_WC`
	`XGL_CURVE_RELATIVE_VDC`
	`XGL_CURVE_RELATIVE_DC`
	See the `XGL_CTX_NURBS_CURVE_APPROX` man page for more information.

Xgl_error_type	Renamed the *Xgl_error_type* error type value `XGL_ERROR_NON_RECOVERABLE to XGL_ERROR_NONRECOVERABLE.`

Xgl_facet_flags	Changed to add new facet flags. The new facet flags are:
	`XGL_FACET_FLAG_SHAPE_UNKNOWN`
	`XGL_FACET_FLAG_DATA_CONTIG`
	`XGL_FACET_FLAG_DATA_NOT_CONTIG`
	See the `xgl_multi_simple_polygon()` man page.

Table B-16 Changed Data Structures *(Continued)*

Data Structure	Change
Xgl_hlhsr_mode	Renamed the hidden line/surface mode value `XGL_HLHSR_ZBUFFER` to `XGL_HLHSR_Z_BUFFER`.
Xgl_inquire	Added new elements to the structure to provide more information on what can be accelerated. For detailed information, see the `xgl_inquire` man page.
Xgl_nurbs_curve	Renamed the *Xgl_nurbs_curve* (XGL 2.0) data structure to *Xgl_nu_bspline_curve* and added a new *Xgl_nurbs_curve* data structure.
Xgl_obj_desc	Added a `stream` structure to the union of descriptive structures required by some objects at object creation. The stream structure is used in the creation of a Stream Device object.
Xgl_plane	Changed the value type of *pt* and *normal* in the *Xgl_plane* data structure from *Xgl_pt_f3d* to *Xgl_pt_d3d*.
Xgl_primitive_type	Removed the value `XGL_PRIM_NU_BSPLINE_CURVE` and added the values `XGL_PRIM_NURBS_CURVE`, `XGL_PRIM_NURBS_SURFACE`, and `XGL_PRIM_TRIANGLE_LIST`.
Xgl_pt_list	Added a new field `num_data_values` to this structure to support texture mapping.
Xgl_pt_pcnf	Renamed the *Xgl_pt_pcnf* data structure to *Xgl_pt_ptr_union* and added double precision point types (d2d and d3d) and data point types to this union of pointers. For detailed information, see the `xgl_pt_list` man page.
Xgl_pt_type	Added double precision point types (d2d and d3d) and data point types to this structure. For detailed information, see *Xgl_pt_type* in the `xgl_enum_types` man page.
Xgl_sys_st	Changed to *Xgl_sys_state*.
Xgl_text_align_horiz	Has been renamed to *Xgl_stext_align_horiz*. The values for the ennumerated type have been changed to `XGL_STEXT_ALIGN_HORIZ_LEFT`, `XGL_STEXT_ALIGN_HORIZ_CENTER`, `XGL_STEXT_ALIGN_HORIZ_RIGHT`, `XGL_STEXT_ALIGN_HORIZ_NORMAL`.

Table B-16 Changed Data Structures *(Continued)*

Data Structure	Change
Xgl_text_align_vert	Has been renamed to *Xgl_stext_align_vert*. The values for the ennumerated type have changed to `XGL_STEXT_ALIGN_VERT_TOP`, `XGL_STEXT_ALIGN_VERT_CAP`, `XGL_STEXT_ALIGN_VERT_HALF`, `XGL_STEXT_ALIGN_VERT_BASE`, `XGL_STEXT_ALIGN_VERT_BOTTOM`, `XGL_STEXT_ALIGN_VERT_NORMAL`.
Xgl_text_path	Has been renamed to *Xgl_stext_path*. The values for the ennumerated type have changed to `XGL_STEXT_PATH_RIGHT`, `XGL_STEXT_PATH_LEFT`, `XGL_STEXT_PATH_UP`, `XGL_STEXT_PATH_DOWN`.
Xgl_text_prec	Has changed to *Xgl_stext_precision*. The only available enumerated type value is now `XGL_STEXT_PRECISION_STROKE`.

Deleted Data Structures

The data structures in Table B-17 have been deleted and are not supported in XGL 3.x.

Table B-17 Deleted Data Structures

Data Structure	Description
Xgl_pt_b2d *Xgl_pt_color_b2d* *Xgl_pt_flag_b2d* *Xgl_pt_b2h* *Xgl_bounds_b2d* *Xgl_arc_b2d* *Xgl_circle_b2d* *Xgl_matrix_b2d* *Xgl_rect_b2d*	Fract point types have been deleted from XGL's 3D pipeline.
Xgl_pt_i3d *Xgl_pt_color_i3d* *Xgl_pt_normal_i3d* *Xgl_pt_color_normal_i3d* *Xgl_pt_flag_i3d* *Xgl_pt_color_flag_i3d* *Xgl_pt_normal_flag_i3d* *Xgl_pt_color_normal_flag_i3d* *Xgl_pt_i3h* *Xgl_matrix_i3d* *Xgl_bounds_i3d*	Integer point types have been deleted from XGL's 2D pipeline.
Xgl_cgm *Xgl_cgm_description* *Xgl_cgm_encoding* *Xgl_cgm_scale_mode* *Xgl_cgm_type* *Xgl_cgm_vdc_extent*	CGM data types have been deleted.

Software Rendering Characteristics

XGL provides a software-only implementation of most primitives and attributes. This software implementation may be used in cases when a hardware device cannot process a primitive or attribute. This appendix provides information on some of the properties and limitations of software-only rendering.

Antialiasing

The following notes list characteristics of software rendering of antialiased primitives.

- For all the antialiasing attributes, the software implementation only handles rendering through a 3D Context. The antialiasing attributes are the following:

  ```
  XGL_CTX_LINE_AA_BLEND_EQ
  XGL_CTX_LINE_AA_FILTER_WIDTH
  XGL_CTX_LINE_AA_FILTER_SHAPE

  XGL_CTX_EDGE_AA_BLEND_EQ
  XGL_CTX_EDGE_AA_FILTER_WIDTH
  XGL_CTX_EDGE_AA_FILTER_SHAPE

  XGL_CTX_MARKER_AA_BLEND_EQ
  XGL_CTX_MARKER_AA_FILTER_WIDTH
  XGL_CTX_MARKER_AA_FILTER_SHAPE
  ```

XGL_CTX_STEXT_AA_BLEND_EQ
XGL_CTX_STEXT_AA_FILTER_WIDTH
XGL_CTX_STEXT_AA_FILTER_SHAPE

XGL_CTX_SURF_AA_BLEND_EQ
XGL_CTX_SURF_AA_FILTER_WIDTH
XGL_CTX_SURF_AA_FILTER_SHAPE

- Software antialiasing works when the XGL_DEV_COLOR_TYPE is set to XGL_COLOR_RGB. When the device color type is XGL_COLOR_INDEX, strokes and dot markers are drawn unantialiased.
- Antialiasing smooths out single-pixel-wide edges, lines, markers, text, hollow surfaces, and dots by spreading out the stroke (or dot) over two adjacent pixels. The blurring of the geometry is done by filtering the primitive's pixels onto neighboring pixels. The width of the filter is controlled by the attribute XGL_CTX_{*primitive_type*}_AA_FILTER_WIDTH. This attribute specifies the number of pixels touched by the filter. A hardware device will use the largest filter width it supports that is less than or equal to the specified filter width. The software implementation of antialiasing allows a maximum value of 3 for XGL_CTX_{*primitive_type*}_AA_FILTER_WIDTH. In other words, if this attribute is set to a value greater than 3 by the application, XGL uses the value of 3 for rendering through software.
- If you are running XGL on a device that does not do gamma correction in hardware, then you can specify the gamma correction value to be used by setting the XGL environment variable XGL_AA_GAMMA_VALUE to a float value between 1.0 and 3.0 (both inclusive). Values will be clamped between 1.0 and 3.0. *Note that this value only affects the visual appearance of antialiased strokes and dot markers drawn in software.* Also, XGL attempts to adjust colors of antialiased primitives such that non-antialiased primitive color is close to the apparent color of the antialiased primitive of the same color.

  Internally XGL uses a value of 2.22 for gamma correction, although this value may change in future releases. Setting the environment variable to a value of 1.0 will result in no change in the visual appearance of antialiased primitives on a device that does gamma correction in hardware.

  For general information on gamma correction, see Foley, van Dam, Feiner, and Hughes, *Computer Graphics: Principles and Practice*, Second Edition, Addison-Wesley Publishing Company, 1990.

- When antialiasing is done in software, antialiased stroke primitives (including edges) and antialiased dot markers look best on a TrueColor visual. On a PseudoColor visual, dithering causes some deterioration in the appearance of antialiased primitives.

Transparency

This section provides information on the software rendering of transparent primitives.

- If the XGL_3D_CTX_SURF_TRANSP_METHOD is XGL_TRANSP_BLENDED and transparency is in effect (which means that the three transparency attributes have values implying transparency) but XGL_DEV_COLOR_TYPE is XGL_COLOR_INDEX, then the transparency method used in the software implementation of transparency is XGL_TRANSP_SCREEN_DOOR.

  If the XGL_CTX_ROP is set to anything other than XGL_ROP_COPY and transparency is in effect, then the attribute XGL_TRANSP_BLENDED is interpreted as XGL_TRANSP_SCREEN_DOOR.
- When blended transparency is done in software, primitives rendered with blended transparency looks best on a TrueColor visual. On a PseudoColor visual, dithering causes some deterioration in the appearance of primitives rendered with blended transparency.

The Utility and Main Example Programs

This appendix lists the utilities program and main program used by all the example programs in this manual.

ex_utils.c

This is the utility file. It creates the display, color table, and various Contexts, and is responsible for the window system interaction of most of the example programs.

```
/*
 * ex_utils.c
 */
/*
#include <xview/xview.h>
#include <xview/frame.h>
#include <xview/panel.h>
#include <xview/textsw.h>
#include <xview/canvas.h>
#include <xview/xv_xrect.h>
#include <X11/Xlib.h>
#include <X11/Xatom.h>
#include <X11/Xutil.h>
#include <xgl/xgl.h>

#include "ex.h"

/* accelerated color type of window raster */
Xgl_color_type       ex_color_type;
```

```
Xgl_sgn32      ex_win_depth;
Xgl_boolean   rgb_only = FALSE;

/* colors used in example programs */
Xgl_color     background_color, red_color, green_color, blue_color;
Xgl_color     yellow_color, cyan_color, magenta_color, white_color;
Xgl_color_rgb    background_rgb =    { 0.2, 0.2, 0.2 };
Xgl_color_rgb    red_rgb =           { 1.0, 0.0, 0.0 };
Xgl_color_rgb    green_rgb =         { 0.0, 1.0, 0.0 };
Xgl_color_rgb    blue_rgb =          { 0.0, 0.0, 1.0 };
Xgl_color_rgb    yellow_rgb =        { 1.0, 1.0, 0.0 };
Xgl_color_rgb    cyan_rgb =          { 0.0, 1.0, 1.0 };
Xgl_color_rgb    magenta_rgb =       { 1.0, 0.0, 1.0 };
Xgl_color_rgb    white_rgb =         { 1.0, 1.0, 1.0 };

Xgl_object      sys_st;
Xgl_obj_desc    obj_desc;
Xgl_X_window    xgl_x_win;

static Frame          frame;
static Frame          subframe;
static Panel_item     example_item;
static Panel_item     desc_item;
static Textsw         src_sw;
static Canvas         canvas;

static int            cur_example_num;
static int            max_example_num;
static int            n_examples;
static Example       *examples;
static Xgl_object     ctx_2d;
static Xgl_object     ctx_3d;
static Xgl_object     ras = NULL;

static int            next_example ();
static int            prev_example ();
static void           set_example_num (Panel_item, int, Event*);
static void           do_example ();
static void           set_labels_and_stuff ();
static void           show_src ();
static void           repaint_proc ();
static void           resize_proc (int, int);
static void           event_proc (Xv_Window, Event*, Notify_arg);
static void           xv_wm_install (Frame, Canvas);

static void           (*pick_func) (Xgl_object, int, int) = NULL;
```

```
Frame
ex_init (
    int                 *argc,
    char                *argv[],
    char                *title,
    int                  n_ex,
    Example             *ex)
{
    Display             *display;
    Panel                panel;
    int                  i, win_visual_class;
    char                 label[80];
    XVisualInfo          visual_info;
    int                  visual_class = TrueColor;
    int                  default_screen;
    int                  match_found = 0;

    /*
     * Base frame and panel setup.
     */

    sprintf (label, "XGL examples -- %s", title);
    xv_init (XV_INIT_ARGC_PTR_ARGV, argc, argv, NULL);

    frame = (Frame) xv_create (NULL, FRAME,
                               FRAME_LABEL, label,
                               XV_WIDTH, 500,
                               XV_HEIGHT, 500,
                               NULL);

    display = (Display *) xv_get (frame, XV_DISPLAY);

    /* locate a visual */
    default_screen = DefaultScreen(display);
   while (!(match_found = XMatchVisualInfo(display, default_screen,
                           24, visual_class, &visual_info)) &&
           (visual_class-- > PseudoColor));

    if (match_found) {
        ex_win_depth = 24;
        win_visual_class = visual_info.class;
    }
    else {
        ex_win_depth = 8;
        win_visual_class = PseudoColor;
```

```
    }

    panel = (Panel) xv_create (frame, PANEL, NULL);

    example_item = xv_create (panel, PANEL_CHOICE_STACK,
                              PANEL_LABEL_STRING, "Example",
                              PANEL_ITEM_X, xv_col (panel, 1),
                              PANEL_ITEM_Y, xv_row (panel, 0),
                              PANEL_NOTIFY_PROC, set_example_num,
                              NULL);

    n_examples = n_ex;
    examples = ex;

    for (i = 0; i < n_examples; i++) {
        xv_set (example_item, PANEL_CHOICE_STRING, i,
                examples[i].title, NULL);
    }

    cur_example_num = 0;
    max_example_num = n_examples;
    xv_set (example_item, PANEL_VALUE, cur_example_num, NULL);

    (void) xv_create (panel, PANEL_BUTTON,
                      PANEL_LABEL_STRING, "Next",
                      PANEL_ITEM_X, xv_col (panel, 1),
                      PANEL_ITEM_Y, xv_row (panel, 1),
                      PANEL_NOTIFY_PROC, next_example,
                      NULL);

    (void) xv_create (panel, PANEL_BUTTON,
                      PANEL_LABEL_STRING, "Previous",
                      PANEL_ITEM_X, xv_col (panel, 8),
                      PANEL_ITEM_Y, xv_row (panel, 1),
                      PANEL_NOTIFY_PROC, prev_example,
                      NULL);

    desc_item = xv_create (panel, PANEL_MESSAGE,
                           PANEL_ITEM_X, xv_col (panel, 1),
                           PANEL_ITEM_Y, xv_row (panel, 2),
                           PANEL_LABEL_STRING,
                               examples[cur_example_num].desc,
                           NULL);

    (void) xv_create (panel, PANEL_BUTTON,
                      PANEL_LABEL_STRING, "Source Code...",
```

```
                    PANEL_ITEM_X, xv_col (panel, 25),
                    PANEL_ITEM_Y, xv_row (panel, 1),
                    PANEL_NOTIFY_PROC, show_src,
                    NULL);

window_fit_height (panel);

/*
 * Sub frame and its textsw.
 */

subframe = (Frame) xv_create (frame, FRAME,
                              NULL);
src_sw = (Textsw) xv_create (subframe, TEXTSW, NULL);

set_labels_and_stuff ();

/*
 * Canvas and XGL
 */
canvas = (Canvas) xv_create (frame, CANVAS,
                          XV_X,                    0,
                          WIN_BELOW,               panel,
                          CANVAS_AUTO_CLEAR,       FALSE,
                          CANVAS_RETAINED,         FALSE,
                          CANVAS_FIXED_IMAGE,      FALSE,
                          CANVAS_REPAINT_PROC,     repaint_proc,
                          CANVAS_RESIZE_PROC,      resize_proc,
                          WIN_DEPTH,               ex_win_depth,
                        XV_VISUAL_CLASS,     win_visual_class,
                          NULL);

xv_wm_install(frame, canvas);

/*
 * get ready to create XGL raster
 */
{
    Window                  frame_window;
    Window                  canvas_window;
    Xv_window               pw;
    Xgl_object              cmap;
    Xgl_inquire             *inq_info;

    display = (Display *) xv_get (frame, XV_DISPLAY);
```

```
pw = (Xv_Window) canvas_paint_window (canvas);
canvas_window = (Window) xv_get (pw, XV_XID);

frame_window = (Window) xv_get (frame, XV_XID);

xv_set (pw,
        WIN_EVENT_PROC, event_proc,
        WIN_NO_EVENTS,
        WIN_CONSUME_EVENTS,
            WIN_RESIZE, WIN_REPAINT, NULL,
        WIN_CONSUME_X_EVENT_MASK,
            StructureNotifyMask, NULL,
        WIN_CONSUME_PICK_EVENTS,
            WIN_MOUSE_BUTTONS, LOC_DRAG, NULL,
        NULL);

xgl_x_win.X_display = (void *) XV_DISPLAY_FROM_WINDOW (pw);
xgl_x_win.X_window = (Xgl_usgn32) canvas_window;
xgl_x_win.X_screen = (int) DefaultScreen (display);

/* wait for window */
sleep (2);

sys_st = xgl_open (NULL);

obj_desc.win_ras.type = XGL_WIN_X;
obj_desc.win_ras.desc = &xgl_x_win;
if (!(inq_info = xgl_inquire(sys_st, &obj_desc))) {
  printf("error in getting inquiry\n");
  exit(1);
}

if (rgb_only) {
    ex_color_type = XGL_COLOR_RGB;
} else {
    if (inq_info->color_type.index)
        ex_color_type = XGL_COLOR_INDEX;
    else if (inq_info->color_type.rgb)
        ex_color_type = XGL_COLOR_RGB;
    else {
        if (win_visual_class == TrueColor)
            ex_color_type = XGL_COLOR_RGB;
        else
            ex_color_type = XGL_COLOR_INDEX;
    }
}
```

```
free (inq_info);

/* if accelerated color type is indexed then
 * create ex color map */
if (ex_color_type == XGL_COLOR_INDEX) {
    Xgl_color               color_table[8];
    Xgl_color_list          cmap_info;

    /* background is a dark gray */
    color_table[BACKGROUND_INDEX].rgb = background_rgb;
    color_table[WHITE_INDEX].rgb = white_rgb;
    color_table[RED_INDEX].rgb = red_rgb;
    color_table[GREEN_INDEX].rgb = green_rgb;
    color_table[BLUE_INDEX].rgb = blue_rgb;
    color_table[YELLOW_INDEX].rgb = yellow_rgb;
    color_table[CYAN_INDEX].rgb = cyan_rgb;
    color_table[MAGENTA_INDEX].rgb = magenta_rgb;

    cmap_info.start_index = 0;
    cmap_info.length = 8;
    cmap_info.colors = color_table;
    cmap = xgl_object_create (sys_st, XGL_CMAP, 0,
                         XGL_CMAP_COLOR_TABLE_SIZE, 8,
                         XGL_CMAP_COLOR_TABLE, &cmap_info,
                         NULL);
}

ras = xgl_object_create(sys_st, XGL_WIN_RAS, &obj_desc,
                        XGL_DEV_COLOR_TYPE, ex_color_type,
                        NULL);

if (ex_color_type == XGL_COLOR_INDEX) {
   /* setup index color values in global color structures */
    xgl_object_set(ras, XGL_RAS_COLOR_MAP, cmap, NULL);
    background_color.index = BACKGROUND_INDEX;
    red_color.index = RED_INDEX;
    green_color.index = GREEN_INDEX;
    blue_color.index = BLUE_INDEX;
    yellow_color.index = YELLOW_INDEX;
    cyan_color.index = CYAN_INDEX;
    magenta_color.index = MAGENTA_INDEX;
    white_color.index = WHITE_INDEX;
}
else {
    /* setup rgb color values in global color structures */
    background_color.rgb = background_rgb;
```

```
            red_color.rgb = red_rgb;
            green_color.rgb = green_rgb;
            blue_color.rgb = blue_rgb;
            yellow_color.rgb = yellow_rgb;
            cyan_color.rgb = cyan_rgb;
            magenta_color.rgb = magenta_rgb;
            white_color.rgb = white_rgb;
        }

        ctx_2d = xgl_object_create(sys_st, XGL_2D_CTX, 0,
                        XGL_CTX_DEVICE, ras,
                        XGL_CTX_DEFERRAL_MODE, XGL_DEFER_ASAP,
                       XGL_CTX_BACKGROUND_COLOR, &background_color,
                        NULL);

        ctx_3d = xgl_object_create(sys_st, XGL_3D_CTX, 0,
                        XGL_CTX_DEVICE, ras,
                        XGL_CTX_DEFERRAL_MODE, XGL_DEFER_ASAP,
                       XGL_CTX_BACKGROUND_COLOR, &background_color,
                        NULL);

    }

    return frame;
}

ex_set_pick_func (void  (*func)())
{
    pick_func = func;
}

ex_main_loop ()
{
    xv_main_loop (frame);
}

static void
set_labels_and_stuff ()
{
    char                header[80];
    Textsw_status       status;

    xv_set (desc_item,
            PANEL_LABEL_STRING, examples[cur_example_num].desc,
            NULL);
```

```
    sprintf (header, "XGL Example Source Code -- %s",
             examples[cur_example_num].src_file);

    xv_set (subframe,
            FRAME_LABEL, header,
            NULL);

    /* textsw_reset(src_sw, 0, 0); */

    xv_set (src_sw,
            TEXTSW_STATUS, &status,
            TEXTSW_FILE, examples[cur_example_num].src_file,
            TEXTSW_BROWSING, TRUE,
            TEXTSW_DISABLE_CD, TRUE,
            TEXTSW_DISABLE_LOAD, TRUE,
            TEXTSW_AGAIN_RECORDING, FALSE,
            TEXTSW_FIRST, 0,
            NULL);


    if (status != TEXTSW_STATUS_OKAY)
        printf ("status = %d\n", status);
}

static int
next_example ()
{
    if (cur_example_num < (max_example_num - 1))
        cur_example_num++;
    else
        cur_example_num = 0;

    xv_set (example_item, PANEL_VALUE, cur_example_num, NULL);

    do_example ();

    return (XV_OK);
}

static int
prev_example ()
{
    if (cur_example_num > 0)
        cur_example_num--;
    else
        cur_example_num = (max_example_num - 1);
```

```
    xv_set (example_item, PANEL_VALUE, cur_example_num, NULL);

    do_example ();

    return (XV_OK);
}

static void
set_example_num (
    Panel_item          item,
    int                 value,
    Event              *event)
{
    cur_example_num = value;

    do_example ();
}

static void
do_example ()
{
    set_labels_and_stuff ();

    if (examples[cur_example_num].dim_2d) {
        xgl_object_set(ctx_2d, XGL_CTX_PLANE_MASK, -1, 0);
        xgl_context_new_frame (ctx_2d);
        (*examples[cur_example_num].draw_proc) (ctx_2d);
    }
    else {
        xgl_object_set(ctx_3d, XGL_CTX_PLANE_MASK, -1, 0);
        xgl_context_new_frame (ctx_3d);
        (*examples[cur_example_num].draw_proc) (ctx_3d);
    }
}

static void
show_src ()
{
    xv_set (subframe, XV_SHOW, TRUE, NULL);
}
```

```
static void
event_proc (
    Xv_Window            window,
    Event               *event,
    Notify_arg           arg)
{
    switch (event_action (event)) {
    case ACTION_SELECT:
        if (!event_is_down (event) && pick_func) {
            if (examples[cur_example_num].dim_2d)
              (*pick_func)(ctx_2d,event_x (event),event_y (event));
            else
              (*pick_func)(ctx_3d, event_x(event),event_y (event));
        }
        break;
    default:
        break;
    }

    if (event->ie_code == WIN_RESIZE)
      resize_proc (((XConfigureEvent *)(event->ie_xevent))->width,
                  ((XConfigureEvent *)(event->ie_xevent))->height);
    else if (event->ie_code == WIN_REPAINT) {
        if (ras) {
            xgl_window_raster_resize(ras);
            repaint_proc();
        }
    }
}

static void
repaint_proc ()
{
    do_example ();
}

static void
resize_proc (
    int          new_width,
    int          new_height)
{
    Xgl_bounds_d2d      dc_2d;
    Xgl_bounds_d3d      dc_3d;
    Xgl_vdc_map         map_2d, map_3d;

    if (ras)
```

```
        xgl_window_raster_resize(ras);

    xgl_object_get(ctx_2d, XGL_CTX_VDC_MAP, &map_2d);
    xgl_object_get(ctx_3d, XGL_CTX_VDC_MAP, &map_3d);

    /* reset dc viewport if mapping is off */
    if (map_2d == XGL_VDC_MAP_OFF) {
        xgl_object_get(ctx_2d, XGL_CTX_DC_VIEWPORT, &dc_2d);
        dc_2d.xmax =  (double)(new_width  - 1);
        dc_2d.ymax =  (double)(new_height - 1);
        xgl_object_set(ctx_2d, XGL_CTX_DC_VIEWPORT, &dc_2d, NULL);
    }
    if (map_3d == XGL_VDC_MAP_OFF) {
        xgl_object_get(ctx_3d, XGL_CTX_DC_VIEWPORT, &dc_3d);
        dc_3d.xmax = (double)(new_width  - 1);
        dc_3d.ymax = (double)(new_height - 1);
        xgl_object_set(ctx_3d, XGL_CTX_DC_VIEWPORT, &dc_3d, NULL);
    }
}

/*
 *     Given a xview frame and canvas; tell the server/window manager
 *      to track the cursor and color map events
 */
static void
xv_wm_install(
    Frame       frame,
    Canvas      canvas)
{
    Atom        catom;
    Window      frame_window, canvas_window;
    Display     *display;

    display       = (Display *)xv_get(frame, XV_DISPLAY);
    canvas_window =
(Window)xv_get((Xv_Window)canvas_paint_window(canvas),
                    XV_XID);

    frame_window  = (Window)xv_get(frame, XV_XID);
   catom        = XInternAtom(display,"WM_COLORMAP_WINDOWS",False);
    XChangeProperty(display, frame_window, catom, XA_WINDOW,
            32, PropModeAppend, (unsigned char *)&canvas_window, 1);
    return;
}
```

color_main.c

```
/*
 * color_main.c
 */
#include <xview/xview.h>
#include <xgl/xgl.h>

#include "ex.h"

extern                  ex_main_loop ();
extern Frame            ex_init (int*, char**, char*, int, Example*);
extern void             color_simple (Xgl_object);
extern void             color_ramp (Xgl_object);
extern void             color_dbuf (Xgl_object);
void                    data_init ();

Example                 exs[] = {
    {
        "Simple", "Creates a simple color map object.",
        "color_simple.c", color_simple, TRUE
    },
    {
        "Ramps", "Creates a color map object with ramps.",
        "color_ramp.c", color_ramp, FALSE
    },
    {
        "Double Buffering",
        "Creates two color map objects appropriate for color map
double buffering.",    "color_dbuf.c", color_dbuf, FALSE
    },
};

int                     n_exs = sizeof (exs) / sizeof (Example);

/* Xgl objects */
Xgl_object              simplecmap,
                        rampcmap,
                        dbufcmap1,
                        dbufcmap2;

#define CUBEVAL .5
/* Geometric data used in examples */
Xgl_pt_f3d              pts_f3d[16] = {
    { CUBEVAL,  CUBEVAL, -CUBEVAL},
    {-CUBEVAL,  CUBEVAL, -CUBEVAL},
```

```
    {-CUBEVAL, -CUBEVAL, -CUBEVAL},
    { CUBEVAL, -CUBEVAL, -CUBEVAL},
    { CUBEVAL,  CUBEVAL, -CUBEVAL},
    { CUBEVAL,  CUBEVAL,  CUBEVAL},
    {-CUBEVAL,  CUBEVAL,  CUBEVAL},
    {-CUBEVAL, -CUBEVAL,  CUBEVAL},
    { CUBEVAL, -CUBEVAL,  CUBEVAL},
    { CUBEVAL,  CUBEVAL,  CUBEVAL},
    { CUBEVAL, -CUBEVAL,  CUBEVAL},
    { CUBEVAL, -CUBEVAL, -CUBEVAL},
    {-CUBEVAL, -CUBEVAL, -CUBEVAL},
    {-CUBEVAL, -CUBEVAL,  CUBEVAL},
};
Xgl_pt_list          pl_3d;

Xgl_rect_i2d         recti2d[256];
Xgl_rect_list        rectlist;

main (
    int                    argc,
    char                  *argv[])
{
    data_init ();

    ex_init (&argc, argv, "Colors", n_exs, exs);

    ex_main_loop ();
}

static
void
data_init ()
{
    pl_3d.pt_type = XGL_PT_F3D;
    pl_3d.bbox = NULL;
    pl_3d.num_pts = sizeof (pts_f3d) / sizeof (Xgl_pt_f3d);
    pl_3d.pts.f3d = pts_f3d;

    simplecmap = rampcmap = dbufcmap1 = dbufcmap2 = NULL;
}
```

copy_raster_main.c

```
/*
 * copy_raster_main.c
 */
#include <xview/xview.h>

#include "ex.h"

extern                  ex_main_loop ();
extern Frame            ex_init (int*, char**, char*, int, Example*);
extern void             copy_raster (Xgl_object);

Example                 exs[] = {
    {
        "Context Copy Raster", "Copy pixels from memory raster to
           window raster.",
        "copy_raster.c", copy_raster, TRUE
    },
};

int                     n_exs = sizeof (exs) / sizeof (Example);

main (
    int                     argc,
    char                   *argv[])
{
    ex_init (&argc, argv, "Context Copy Raster", n_exs, exs);

    ex_main_loop ();
}
```

dcue_main.c

```
/*
 * dcue_main.c
 */
#include <xview/xview.h>
#include <xgl/xgl.h>

#include "ex.h"

extern                  ex_main_loop ();
extern Frame            ex_init (int*, char**, char*, int, Example*);
extern void             dcue_linear (Xgl_object);
```

```
extern void           dcue_scaled (Xgl_object);
void                  data_init ();

Example               exs[] = {
    {
        "Linear", "Displays a cube using linear depth cueing.",
        "dcue_linear.c", dcue_linear, FALSE
    },
    {
        "Scaled", "Displays a cube using scaled depth cueing.",
        "dcue_scaled.c", dcue_scaled, FALSE
    },
};

int                   n_exs = sizeof (exs) / sizeof (Example);

/* Xgl objects */
Xgl_object            rampcmap;

/* Geometric data used in examples */
#define CUBEVAL .5
Xgl_pt_f3d            pts_f3d[16] = {
    { CUBEVAL,  CUBEVAL, -CUBEVAL},
    {-CUBEVAL,  CUBEVAL, -CUBEVAL},
    {-CUBEVAL, -CUBEVAL, -CUBEVAL},
    { CUBEVAL, -CUBEVAL, -CUBEVAL},
    { CUBEVAL,  CUBEVAL, -CUBEVAL},
    { CUBEVAL,  CUBEVAL,  CUBEVAL},
    {-CUBEVAL,  CUBEVAL,  CUBEVAL},
    {-CUBEVAL, -CUBEVAL,  CUBEVAL},
    { CUBEVAL, -CUBEVAL,  CUBEVAL},
    { CUBEVAL,  CUBEVAL,  CUBEVAL},
    { CUBEVAL, -CUBEVAL,  CUBEVAL},
    { CUBEVAL, -CUBEVAL, -CUBEVAL},
    {-CUBEVAL, -CUBEVAL, -CUBEVAL},
    {-CUBEVAL, -CUBEVAL,  CUBEVAL},
    {-CUBEVAL,  CUBEVAL,  CUBEVAL},
    {-CUBEVAL,  CUBEVAL, -CUBEVAL},
};
Xgl_pt_list           pl_3d;

main (
    int                   argc,
    char                 *argv[])
{
    data_init ();
```

```
    ex_init (&argc, argv, "Depth Cueing", n_exs, exs);

    ex_main_loop ();
}

static
void
data_init ()
{
    pl_3d.pt_type = XGL_PT_F3D;
    pl_3d.bbox = NULL;
    pl_3d.num_pts = sizeof (pts_f3d) / sizeof (Xgl_pt_f3d);
    pl_3d.pts.f3d = pts_f3d;

    rampcmap = NULL;
}
```

gcache_main.c

```
/*
 * gcache_main.c
 */
#include <xview/xview.h>
#include "ex.h"

extern          ex_main_loop();
extern Frame    ex_init(int*, char**, char*, int, Example*);
extern void     gcache_nsi_pgon(Xgl_object),
gcache_complex_pgon(Xgl_object);

Example         exs[] = {
  {
    "NSI polygon",
    "Shows non self-intersecting polygon and gcache polygon
         of similar shape",
    "gcache_nsi_pgon.c", gcache_nsi_pgon, FALSE
  },
  {
    "Complex polygon",
    "Shows complex self-intersecting polygon and gcache polygon of
          similar shape",
    "gcache_complex_pgon.c", gcache_complex_pgon, FALSE
  }
};
```

```
int             n_exs = sizeof(exs) / sizeof(Example);

main(
  int              argc,
  char            *argv[])
{
  ex_init(&argc, argv, "Gcache", n_exs, exs);
  ex_main_loop();
}
```

gspixel_main.c

```
/*
 * gspixel_main.c
 */
#include <xview/xview.h>
#include <xgl/xgl.h>

#include "ex.h"

extern          ex_main_loop ();
extern  Frame   ex_init (int*, char**, char*, int, Example*);
extern  void    gspixel (Xgl_object);

Example              exs[] = {
    {
        "Get/Set Pixel", "Get and set pixels.",
        "gspixel.c", gspixel, TRUE
    },
};

int                  n_exs = sizeof (exs) / sizeof (Example);

main (
    int                  argc,
    char                *argv[])
{
    ex_init (&argc, argv, "Get/Set Pixel", n_exs, exs);

    ex_main_loop ();
}
```

light_main.c

```
/*
 * light_main.c
 */
#include <xview/xview.h>
#include <xgl/xgl.h>
#include "ex.h"
#include "light.h"

extern                  ex_main_loop ();
extern Frame            ex_init (int*, char**, char*, int, Example*);
extern void             light_facet (Xgl_object);
extern void             light_vertex (Xgl_object);
void                    data_init ();

Example                 exs[] = {
    {
        "Facet", "Draws an icosahedron with facet illumination.",
        "light_facet.c", light_facet, FALSE
    },
    {
        "Vertex", "Draws an icosahedron with vertex illumination.",
        "light_vertex.c", light_vertex, FALSE
    },
};

int                     n_exs = sizeof (exs) / sizeof (Example);

/* Geometric data used in examples */
Xgl_pt_list             pl[NUM_FACETS];
static Xgl_pt_normal_f3d        pts[NUM_V_PER_F * NUM_FACETS];
static Xgl_pt_f3d       vertex[NUM_VERTICES] =
    {{ 0.00000000,  0.00000000, -0.95105650},
     { 0.00000000,  0.85065080, -0.42532537},
     { 0.80901698,  0.26286556, -0.42532537},
     { 0.50000000, -0.68819095, -0.42532537},
     {-0.50000000, -0.68819095, -0.42532537},
     {-0.80901698,  0.26286556, -0.42532537},
     { 0.50000000,  0.68819095,  0.42532537},
     { 0.80901698, -0.26286556,  0.42532537},
     { 0.00000000, -0.85065080,  0.42532537},
     {-0.80901698, -0.26286556,  0.42532537},
     {-0.50000000,  0.68819095,  0.42532537},
     { 0.00000000,  0.00000000,  0.95105650}};
static Xgl_usgn32       facet[NUM_FACETS][NUM_V_PER_F] =
```

```
    {{0, 1, 2}, {0, 2, 3}, {0, 3, 4}, {0, 4, 5},
     {0, 5, 1}, {2, 1, 6}, {3, 2, 7}, {4, 3, 8},
     {5, 4, 9}, {1, 5, 10}, {2, 6, 7}, {3, 7, 8},
     {4, 8, 9}, {5, 9, 10}, {1, 10, 6}, {11, 7, 6},
     {11, 8, 7}, {11, 9, 8}, {11, 10, 9}, {11, 6, 10}};

main (
    int                     argc,
    char                   *argv[])
{
    data_init ();
    ex_init (&argc, argv, "Lights", n_exs, exs);
    ex_main_loop ();
}

void
data_init ()
{
    Xgl_usgn32              i,
                            j;

    /* For each facet ... */
    for (i = 0; i < NUM_FACETS; i++) {

        /* Initialize the point list structure for current facet */
        pl[i].pt_type = XGL_PT_NORMAL_F3D;
        pl[i].bbox = NULL;
        pl[i].num_pts = NUM_V_PER_F;
        pl[i].pts.normal_f3d = &(pts[NUM_V_PER_F * i]);

        /* For each vertex in this facet ... */
        for (j = 0; j < NUM_V_PER_F; j++) {

            /* Read vertex coordinates */
            pts[NUM_V_PER_F * i + j].x = vertex[facet[i][j]].x;
            pts[NUM_V_PER_F * i + j].y = vertex[facet[i][j]].y;
            pts[NUM_V_PER_F * i + j].z = vertex[facet[i][j]].z;
         pts[NUM_V_PER_F * i + j].normal.x = vertex[facet[i][j]].x /
                                                  0.95105650;
         pts[NUM_V_PER_F * i + j].normal.y = vertex[facet[i][j]].y /
                                                  0.95105650;
         pts[NUM_V_PER_F * i + j].normal.z = vertex[facet[i][j]].z /
                                                  0.95105650;
        }
    }
}
```

lpat_main.c

```
/*
 * lpat_main.c
 */
#include <xview/xview.h>
#include <xgl/xgl.h>

#include "ex.h"

extern                  ex_main_loop ();
extern Frame            ex_init (int*, char**, char*, int, Example*);
extern void             lpat_lines (Xgl_object);
extern void             lpat_pgon_edges (Xgl_object);

Example                 exs[] = {
    {
        "Patterned Lines", "Draws lines using patterns.",
        "lpat_lines.c", lpat_lines, TRUE
    },
    {
        "Patterned Edges", "Draws polygons with patterned edges.",
        "lpat_pgon_edges.c", lpat_pgon_edges, TRUE
    },
};

int                     n_exs = sizeof (exs) / sizeof (Example);

/* Geometric data used in examples */
/* for polygons with pattern edges - make some stars */
#define OFFSET  0
Xgl_pt_i2d              pts_i2d_pgon1[6] = {
    OFFSET, OFFSET, OFFSET + 100, OFFSET + 50,
    OFFSET, OFFSET + 50, OFFSET + 100, OFFSET,
    OFFSET + 50, OFFSET + 100, OFFSET, OFFSET
};

#undef OFFSET
#define OFFSET  100
Xgl_pt_i2d              pts_i2d_pgon2[6] = {
    OFFSET, OFFSET, OFFSET + 100, OFFSET + 50,
    OFFSET, OFFSET + 50, OFFSET + 100, OFFSET,
    OFFSET + 50, OFFSET + 100, OFFSET, OFFSET
};

#undef OFFSET
```

```
#define OFFSET  200
Xgl_pt_i2d          pts_i2d_pgon3[6] = {
    OFFSET, OFFSET, OFFSET + 100, OFFSET + 50,
    OFFSET, OFFSET + 50, OFFSET + 100, OFFSET,
    OFFSET + 50, OFFSET + 100, OFFSET, OFFSET
};

#undef OFFSET
#define OFFSET  300
Xgl_pt_i2d          pts_i2d_pgon4[6] = {
    OFFSET, OFFSET, OFFSET + 100, OFFSET + 50,
    OFFSET, OFFSET + 50, OFFSET + 100, OFFSET,
    OFFSET + 50, OFFSET + 100, OFFSET, OFFSET
};

main (
    int                  argc,
    char                *argv[])
{
    ex_init (&argc, argv, "Line Patterns", n_exs, exs);

    ex_main_loop ();
}
```

nurbs_main.c

```
/*
 * nurbs_main.c
 */
#include <xview/xview.h>
#include <xgl/xgl.h>

#include "ex.h"

extern              ex_main_loop ();
extern Frame        ex_init (int*, char**, char*, int, Example*);
extern void         nurbs_circle (Xgl_object);
extern void         nurbs_bezier (Xgl_object);
extern void         nurbs_sphere (Xgl_object);

Example             exs[] = {
    {
        "Bezier curve", "Display a Bezier curve, using NURBS",
        "nurbs_bezier.c", nurbs_bezier, TRUE
    },
```

```
    {
        "Circles", "Display circles as NURBS curves, using
         different approx criteria",
        "nurbs_circle.c", nurbs_circle, TRUE
    },
    {
       "Trimmed Sphere", "Display a trimmed sphere using NURBS
    surface", "nurbs_sphere.c", nurbs_sphere, FALSE
    }
};

int                     n_exs = sizeof (exs) / sizeof (Example);

main (
    int                     argc,
    char                   *argv[])
{

    ex_init (&argc, argv, "NURBS", n_exs, exs);

    ex_main_loop ();
}
```

pick_2d_main.c

```
/*
 * pick_2d_main.c
 */

#include <xview/xview.h>
#include "ex.h"

extern                  ex_main_loop ();
extern Frame            ex_init (int*, char**, char*, int, Example*);
extern void             ex_set_pick_func(void ((*)(Xgl_object,
                                        int, int)));
extern void             pick_draw_scene (Xgl_object);
extern void             pick_do_pick (Xgl_object, int, int);

Example                 exs[] = {
    {
        "2d Picking", "Pick primitives in a scene",
        "pick_2d_prims.c", pick_draw_scene, TRUE
    }
};
```

```
int                  n_exs = sizeof (exs) / sizeof (Example);

main (
    int                  argc,
    char                *argv[])
{
    ex_init (&argc, argv, "2d Picking", n_exs, exs);
    ex_set_pick_func (pick_do_pick);
    ex_main_loop ();
}
```

prims_2d_main.c

```
/*
 * prims_2d_main.c
 */
#include <xview/xview.h>
#include "ex.h"

extern               ex_main_loop ();
extern Frame         ex_init (int*, char**, char*, int, Example*);
extern void          prims_2d_pgon (Xgl_object);
extern void          prims_2d_marker (Xgl_object);
extern void          prims_2d_umarker (Xgl_object);
extern void          prims_2d_rect (Xgl_object);
extern void          prims_2d_circle (Xgl_object);

Example              exs[] = {
    {
        "Polygon", "Draws a polygon with a hole.",
        "prims_2d_pgon.c", prims_2d_pgon, TRUE
    },
    {
        "Rectangles", "Draws rectangles.",
        "prims_2d_rect.c", prims_2d_rect, TRUE
    },
    {
        "Circles", "Draws circles.",
        "prims_2d_circle.c", prims_2d_circle, TRUE
    },
    {
        "Marker", "Draws markers.",
        "prims_2d_marker.c", prims_2d_marker, TRUE
    },
```

```
    {
        "Markers_II", "Draws your marker.",
        "prims_2d_umarker.c", prims_2d_umarker, TRUE
    },
};

int                    n_exs = sizeof (exs) / sizeof (Example);

main (
    int                    argc,
    char                  *argv[])
{
    ex_init (&argc, argv, "2d Primitives", n_exs, exs);
    ex_main_loop ();
}
```

tran_main.c

```
/*
 * tran_main.c
 */

#include <xview/xview.h>
#include <xgl/xgl.h>

#include "ex.h"

extern                 ex_main_loop ();
extern Frame           ex_init (int*, char**, char*, int, Example*);
extern void            tran_2d_orig (Xgl_object);
extern void            tran_2d_translate (Xgl_object);
extern void            tran_2d_rotate (Xgl_object);
extern void            tran_2d_scale (Xgl_object);
extern void            tran_3d (Xgl_object);
void                   data_init ();

Example                exs[] = {
    {
        "2D Original", "Draws an untransformed triangle.",
        "tran_2d_orig.c", tran_2d_orig, TRUE
    },
    {
        "2D translation", "Draws a translated triangle.",
        "tran_2d_transl.c", tran_2d_translate, TRUE
    },
```

```
    {
        "2D rotation", "Draws a rotated triangle.",
        "tran_2d_rot.c", tran_2d_rotate, TRUE
    },
    {
        "2D scaling", "Draws a scaled triangle.",
        "tran_2d_scale.c", tran_2d_scale, TRUE
    },
    {
        "3D transformation",
        "Draws a wire-frame cube with a 3D view transform.",
        "tran_3d.c", tran_3d, FALSE
    },
};

int                     n_exs = sizeof (exs) / sizeof (Example);

/* Geometric data used in examples */
Xgl_pt_list             pl_2d;
static Xgl_pt_f2d       pts_f2d[3] =
    {{20.0, 0.0}, {60.0, 0.0}, {40.0, 100.0}};
Xgl_pt_list             pl_3d[4];
static Xgl_pt_f3d       pts_f3d[16] =
    {{ 1.0,  1.0,  1.0}, { 1.0, -1.0,  1.0},
     { 1.0, -1.0, -1.0}, { 1.0,  1.0, -1.0},
     { 1.0,  1.0,  1.0}, {-1.0,  1.0,  1.0},
     {-1.0,  1.0, -1.0}, { 1.0,  1.0, -1.0},
     {-1.0,  1.0,  1.0}, {-1.0, -1.0,  1.0},
     {-1.0, -1.0, -1.0}, {-1.0,  1.0, -1.0},
     { 1.0, -1.0, -1.0}, {-1.0, -1.0, -1.0},
     { 1.0, -1.0,  1.0}, {-1.0, -1.0,  1.0}};

main (
    int                     argc,
    char                   *argv[])
{
    data_init ();
    ex_init (&argc, argv, "Transforms", n_exs, exs);
    ex_main_loop ();
}

void
data_init ()
{
    pl_2d.pt_type = XGL_PT_F2D;
    pl_2d.bbox = NULL;
```

```
    pl_2d.num_pts = 3;
    pl_2d.pts.f2d = pts_f2d;

    pl_3d[0].pt_type = XGL_PT_F3D;
    pl_3d[0].bbox = NULL;
    pl_3d[0].num_pts = 8;
    pl_3d[0].pts.f3d = &(pts_f3d[0]);

    pl_3d[1].pt_type = XGL_PT_F3D;
    pl_3d[1].bbox = NULL;
    pl_3d[1].num_pts = 4;
    pl_3d[1].pts.f3d = &(pts_f3d[8]);

    pl_3d[2].pt_type = XGL_PT_F3D;
    pl_3d[2].bbox = NULL;
    pl_3d[2].num_pts = 2;
    pl_3d[2].pts.f3d = &(pts_f3d[12]);

    pl_3d[3].pt_type = XGL_PT_F3D;
    pl_3d[3].bbox = NULL;
    pl_3d[3].num_pts = 2;
    pl_3d[3].pts.f3d = &(pts_f3d[14]);
}
```

view_main.c

```
/*
 * view_main.c
 */
#include <xview/xview.h>
#include <xview/frame.h>
#include <xview/panel.h>
#include <xgl/xgl.h>

#include "ex.h"

extern                  ex_main_loop ();
extern Frame            ex_init (int*, char**, char*, int, Example*);
extern void             view_perspective (Xgl_object);
extern void             view_set (Panel_item, int, Event*);
extern void             vdc_map_set (Panel_item, int, Event*);

static void             view_panel_set (Frame);
static void             data_init_rgb ();
static void             data_init_index ();
```

```
Frame                   view_frame;
Panel_item              eye_x_item,
                        eye_y_item,
                        eye_z_item,
                        field_of_view_item;

Xgl_pt_list             pls[4];
Xgl_pt_color_f3d        pts[16];

Example                 exs[] = {
    {
      "Perspective",
      "Draws a cube with the specified viewing parameters.",
      "view_perspect.c", view_perspective, FALSE
    },
};

int                     n_exs = sizeof (exs) / sizeof (Example);

/****
 *
 * main
 *
 * Initialize geometric data, windows, XGL, and view panel,
 * then enter event loop.
 *
 ***/
main (
    int                     argc,
    char                   *argv[])
{
    Frame                   frame;


    frame = ex_init (&argc, argv, "View Model", n_exs, exs);

    view_panel_set (frame);

    if (ex_color_type == XGL_COLOR_INDEX)
        data_init_index ();
    else if (ex_color_type == XGL_COLOR_RGB)
        data_init_rgb ();
    else {
        printf("unknown color type\n");
        exit(1);
```

```
    }

    ex_main_loop ();
}

#define         PT_DEF_RGB(ii, xx, yy, zz, cc) {  \
    pts[(ii)].x     = (xx);                       \
    pts[(ii)].y     = (yy);                       \
    pts[(ii)].z     = (zz);                       \
    pts[(ii)].color.rgb = (cc);                   \
}

#define         PT_DEF_INX(ii, xx, yy, zz, cc) {  \
    pts[(ii)].x     = (xx);                       \
    pts[(ii)].y     = (yy);                       \
    pts[(ii)].z     = (zz);                       \
    pts[(ii)].color.index = (cc);                 \
}

/****
 *
 * data_init_rgb
 *
 * Initialize geometric data for a cube composed of cyan,
 * blue, yellow,and white lines.
 *
 * For RGB color type.
 *
 ***/
static void
data_init_rgb ()
{
    /* Set up cube as polylines */
    pls[0].num_pts = 8;
    pls[0].pt_type = XGL_PT_COLOR_F3D;
    pls[0].bbox = NULL;
    pls[0].pts.color_f3d = &(pts[0]);
    PT_DEF_RGB (0, 1.0, 1.0, 1.0, cyan_rgb)
    PT_DEF_RGB (1, 1.0, -1.0, 1.0, cyan_rgb)
    PT_DEF_RGB (2, 1.0, -1.0, -1.0, cyan_rgb)
    PT_DEF_RGB (3, 1.0, 1.0, -1.0, cyan_rgb)
    PT_DEF_RGB (4, 1.0, 1.0, 1.0, cyan_rgb)
    PT_DEF_RGB (5, -1.0, 1.0, 1.0, white_rgb)
    PT_DEF_RGB (6, -1.0, 1.0, -1.0, yellow_rgb)
    PT_DEF_RGB (7, 1.0, 1.0, -1.0, white_rgb)
    pls[1].num_pts = 4;
```

```
    pls[1].pt_type = XGL_PT_COLOR_F3D;
    pls[1].bbox = NULL;
    pls[1].pts.color_f3d = &(pts[8]);
    PT_DEF_RGB (8, -1.0, 1.0, 1.0, yellow_rgb)
    PT_DEF_RGB (9, -1.0, -1.0, 1.0, yellow_rgb)
    PT_DEF_RGB (10, -1.0, -1.0, -1.0, yellow_rgb)
    PT_DEF_RGB (11, -1.0, 1.0, -1.0, yellow_rgb)
    pls[2].num_pts = 2;
    pls[2].pt_type = XGL_PT_COLOR_F3D;
    pls[2].bbox = NULL;
    pls[2].pts.color_f3d = &(pts[12]);
    PT_DEF_RGB (12, 1.0, -1.0, -1.0, blue_rgb)
    PT_DEF_RGB (13, -1.0, -1.0, -1.0, blue_rgb)
    pls[3].num_pts = 2;
    pls[3].pt_type = XGL_PT_COLOR_F3D;
    pls[3].bbox = NULL;
    pls[3].pts.color_f3d = &(pts[14]);
    PT_DEF_RGB (14, 1.0, -1.0, 1.0, blue_rgb)
    PT_DEF_RGB (15, -1.0, -1.0, 1.0, blue_rgb)
}

/****
 *
 * data_init_index
 *
 * Initialize geometric data for a cube composed of
 * cyan, blue, yellow,and white lines.
 *
 * For Indexed color type.
 *
 ***/
static void
data_init_index ()
{
    /* Set up cube as polylines */
    pls[0].num_pts = 8;
    pls[0].pt_type = XGL_PT_COLOR_F3D;
    pls[0].bbox = NULL;
    pls[0].pts.color_f3d = &(pts[0]);
    PT_DEF_INX (0, 1.0, 1.0, 1.0, CYAN_INDEX)
    PT_DEF_INX (1, 1.0, -1.0, 1.0, CYAN_INDEX)
    PT_DEF_INX (2, 1.0, -1.0, -1.0, CYAN_INDEX)
    PT_DEF_INX (3, 1.0, 1.0, -1.0, CYAN_INDEX)
    PT_DEF_INX (4, 1.0, 1.0, 1.0, CYAN_INDEX)
    PT_DEF_INX (5, -1.0, 1.0, 1.0, WHITE_INDEX)
    PT_DEF_INX (6, -1.0, 1.0, -1.0, YELLOW_INDEX)
```

```
    PT_DEF_INX (7, 1.0, 1.0, -1.0, WHITE_INDEX)
    pls[1].num_pts = 4;
    pls[1].pt_type = XGL_PT_COLOR_F3D;
    pls[1].bbox = NULL;
    pls[1].pts.color_f3d = &(pts[8]);
    PT_DEF_INX (8, -1.0, 1.0, 1.0, YELLOW_INDEX)
    PT_DEF_INX (9, -1.0, -1.0, 1.0, YELLOW_INDEX)
    PT_DEF_INX (10, -1.0, -1.0, -1.0, YELLOW_INDEX)
    PT_DEF_INX (11, -1.0, 1.0, -1.0, YELLOW_INDEX)
    pls[2].num_pts = 2;
    pls[2].pt_type = XGL_PT_COLOR_F3D;
    pls[2].bbox = NULL;
    pls[2].pts.color_f3d = &(pts[12]);
    PT_DEF_INX (12, 1.0, -1.0, -1.0, BLUE_INDEX)
    PT_DEF_INX (13, -1.0, -1.0, -1.0, BLUE_INDEX)
    pls[3].num_pts = 2;
    pls[3].pt_type = XGL_PT_COLOR_F3D;
    pls[3].bbox = NULL;
    pls[3].pts.color_f3d = &(pts[14]);
    PT_DEF_INX (14, 1.0, -1.0, 1.0, BLUE_INDEX)
    PT_DEF_INX (15, -1.0, -1.0, 1.0, BLUE_INDEX)
}

/****
 *
 * view_panel_set
 *
 * Create and initialize a separate panel for entering view
 * model parameters.
 *
 ***/
static
void
view_panel_set (frame)
    Frame                frame;
{
    Panel                panel;

    view_frame = (Frame) xv_create (frame, FRAME,
                           FRAME_LABEL, "View Model Parameters",
                           NULL);

    panel = (Panel) xv_create (view_frame, PANEL, NULL);

    panel_create_item (panel, PANEL_MESSAGE,
                       PANEL_LABEL_STRING,
```

```
                              "Resize the window to see the effect of
                                 VDC mapping.",
                              PANEL_ITEM_X, xv_col (panel, 2),
                              PANEL_ITEM_Y, xv_row (panel, 4),
                              NULL);
(void) panel_create_item (panel, PANEL_CYCLE,
                              PANEL_LABEL_STRING, "VDC map:",
                          PANEL_CHOICE_STRINGS, "All", "Aspect", 0,
                              PANEL_VALUE, 1,
                              PANEL_NOTIFY_PROC, vdc_map_set,
                              PANEL_ITEM_X, xv_col (panel, 2),
                              PANEL_ITEM_Y, xv_row (panel, 0),
                              NULL);
eye_x_item = panel_create_item (panel, PANEL_TEXT,
                              PANEL_NOTIFY_LEVEL, PANEL_NONE,
                              PANEL_VALUE_DISPLAY_LENGTH, 6,
                              PANEL_ITEM_X, xv_col (panel, 2),
                              PANEL_ITEM_Y, xv_row (panel, 1),
                             PANEL_LABEL_STRING, "Eye position x:",
                              PANEL_VALUE, "6.0",
                              NULL);

eye_y_item = panel_create_item (panel, PANEL_TEXT,
                              PANEL_NOTIFY_LEVEL, PANEL_NONE,
                              PANEL_VALUE_DISPLAY_LENGTH, 6,
                              PANEL_ITEM_X, xv_col (panel, 2),
                              PANEL_ITEM_Y, xv_row (panel, 2),
                             PANEL_LABEL_STRING, "Eye position y:",
                              PANEL_VALUE, "8.0",
                              NULL);

eye_z_item = panel_create_item (panel, PANEL_TEXT,
                              PANEL_NOTIFY_LEVEL, PANEL_NONE,
                              PANEL_VALUE_DISPLAY_LENGTH, 6,
                              PANEL_ITEM_X, xv_col (panel, 2),
                              PANEL_ITEM_Y, xv_row (panel, 1),
                             PANEL_LABEL_STRING, "Eye position x:",
                              PANEL_VALUE, "6.0",
                              NULL);

eye_y_item = panel_create_item (panel, PANEL_TEXT,
                              PANEL_NOTIFY_LEVEL, PANEL_NONE,
                              PANEL_VALUE_DISPLAY_LENGTH, 6,
                              PANEL_ITEM_X, xv_col (panel, 2),
                              PANEL_ITEM_Y, xv_row (panel, 2),
                             PANEL_LABEL_STRING, "Eye position y:",
```

```
                              PANEL_VALUE, "8.0",
                              NULL);

    eye_z_item = panel_create_item (panel, PANEL_TEXT,
                              PANEL_NOTIFY_LEVEL, PANEL_NONE,
                              PANEL_VALUE_DISPLAY_LENGTH, 6,
                              PANEL_ITEM_X, xv_col (panel, 2),
                              PANEL_ITEM_Y, xv_row (panel, 3),
                             PANEL_LABEL_STRING, "Eye position z:",
                              PANEL_VALUE, "7.5",
                              NULL);

    (void) panel_create_item (panel, PANEL_BUTTON,
                              PANEL_ITEM_X, xv_col (panel, 30),
                              PANEL_ITEM_Y, xv_row (panel, 1),
                              PANEL_NOTIFY_PROC, view_set,
                              PANEL_LABEL_STRING, "Set view",
                              NULL);

    field_of_view_item = panel_create_item (panel, PANEL_TEXT,
                              PANEL_NOTIFY_LEVEL, PANEL_NONE,
                              PANEL_VALUE_DISPLAY_LENGTH, 6,
                              PANEL_ITEM_X, xv_col (panel, 26),
                              PANEL_ITEM_Y, xv_row (panel, 3),
                             PANEL_LABEL_STRING,"X field of view:",
                              PANEL_VALUE, "20.0",
                              NULL);

    window_fit (panel);
    window_fit (view_frame);
    xv_set (view_frame, XV_SHOW, TRUE, NULL);
}
```

XGL Errors

This appendix lists XGL errors. The error number and error message are given for each error message that XGL defines. An error is *NONRECOVERABLE* if XGL aborts further processing and returns control back to the caller. Errors of this type would cause a *core dump* if the application were allowed to continue execution, because these errors leave XGL in an inconsistent state. An error is *RECOVERABLE* if XGL can make reasonable assumptions about what the application intended to do and then continue processing. For instance, the library will usually use the default value for an incorrect parameter, and then continue the application.

The application can filter XGL errors by specifying its own error reporting function with the `XGL_SYS_ST_ERROR_NOTIFICATION_FUNCTION` attribute. For more information on XGL error detection and reporting, see“Error Detection and Reporting” on page 45.

Device-Independent Nonrecoverable Errors

di-1	Malloc or new failed: out of memory
di-2	Internal error
di-3	Primitive, attribute or object not (yet) implemented
di-4	Primitive cannot be called at this time
di-5	Cannot load font

di-6 Unable to load loadable pipeline

di-7 Raster depth/data type not supported

di-8 Raster not created

di-9 Cannot obtain graphics hardware resource

di-10 Notify_handler: unknown msg->type

di-11 xgl_obj_internal_get: getting undefined internal attribute

di-12 xgl_obj_set: setting undefined internal attribute

di-13 Cannot read error file

di-14 Unsupported window type

di-15 wx_grab: cannot open

di-16 wx_grab: cannot map lock page

di-17 wx_grab: cannot map unlock page

di-18 xgl_obj_free: object still used; cannot free

di-19 Can't map framebuffer memory

di-20 Can't unmap framebuffer memory

di-21 Can't determine framebuffer type

di-22 Can't open framebuffer; unsupported framebuffer type

di-23 Unexpected EOF in font description file

di-24 Unexpected framebuffer type

di-25 Could not obtain the hardware resource of window ids.

di-26 DGA Protocol initialization failed

di-27 Unknown visual class.

di-28 Bad magic number in font description file

di-30 No Color Map object

di-31 Dither mask not defined

di-32	Model clipping memory allocation failed
di-33	Cannot create Window Raster with requested protocol
di-34	Invalid protocol or multiple protocols requested
di-35	Invalid Memory Raster depth; must use 8 or 32
di-36	Can't create System State
di-37	Can't create System State - Unable to create Predefined Objects
di-38	Context initialization failed
di-39	Object creation failed
di-40	LI-3 function not implemented by device pipeline
di-41	xgl.modules file not found
di-42	Requested pipeline not found in xgl.modules
di-43	Bad line in xgl.modules
di-44	Unknown error reading xgl.modules

Device-Independent Recoverable Errors

di-102	Invalid parameter for attribute-value pair
di-103	No XGL_CMAP_COLOR_MAPPER; i.e. no mapping from RGB to Index
di-104	No XGL_CMAP_INVERSE_COLOR_MAPPER
di-106	Invalid Context
di-108	Color map size reduced to the color table size
di-109	Too many colors
di-110	Dither mask memory error
di-111	Invalid destination object
di-112	Invalid source object

di-113	Invalid color map
di-114	Invalid address for colors
di-115	Attribute is read-only
di-120	Object is read-only
di-121	Different/mixed dimensions or depths
di-122	Invalid value
di-123	Not available for this type:
di-126	Attribute not defined for object
di-127	Invalid parameter for operator
di-128	Singular transform
di-129	Cannot set attribute(s) of predefined line patterns
di-130	Order is not supported
di-131	Invalid address for curve
di-132	Invalid address for knot_vector
di-133	Invalid address for control points
di-134	Incorrect number of knots/control points
di-135	Incorrect point type for nurbs
di-136	Unknown class
di-137	Invalid address for circle/arc data
di-138	Invalid address for string
di-139	Invalid address for position/point/point-list
di-141	Invalid values in Up Vector: cannot handle x = y = 0
di-142	Invalid address for direction vectors
di-143	Invalid address for rectangle
di-144	Invalid address for concatenation point

di-145	Invalid or unsupported value for attribute
di-147	Invalid point or facet type
di-148	Invalid address for matrix
di-149	Invalid axis value
di-150	Invalid update value
di-151	Invalid address for scale factors
di-152	Invalid address for offset
di-153	Named color map
di-154	Can't match PseudoColor visual
di-155	Can't allocate X colors
di-157	Not a Memory Raster; can't set/get address
di-158	Cannot change attribute; it is writable only once
di-159	Color cube dimensions too large; using defaults
di-160	Color table too small
di-161	Cannot reallocate clip_mask
di-162	Cannot reallocate Z-Buffer
di-167	Value out of range for attribute
di-168	Cannot change pick style when picking enabled
di-169	Cannot change pick buffer size when picking enabled
di-170	Invalid or inconsistent data for primitive
di-171	No Device attached to Context
di-172	Cannot pop Context: stack is empty
di-173	Invalid function for this class
di-174	Cannot execute operator, no System state
di-175	Invalid flags data

di-176	Invalid address, type, or insufficient facet information
di-178	Illegal clipping code
di-179	Invalid values of row/column
di-180	Invalid address for color
di-181	Degenerate facet: cannot compute normal. Using {1, 0, 0} as normal
di-183	Invalid address
di-185	Failed to close polygon edge
di-186	Invalid pattern
di-198	System State already exists; xgl_open can be called only once
di-201	Cannot compute facet normal: all points are at same position
di-202	Cannot compute facet normal: all points are colinear
di-203	Context not supported by this Gcache operation
di-204	Cannot use a Gcache with an annotation ellipse
di-206	Front reference plane is behind back reference plane
di-207	Cannot push matrix
di-208	Cannot pop matrix
di-209	Stack too small
di-210	Stack empty
di-212	Insufficient memory for copying primitive data; size follows:
di-213	Invalid octant encountered when creating edge lists
di-216	More buffers requested than hardware supports
di-217	Invalid model clip update method
di-218	Invalid point type or dimension in this Context
di-219	Can't find object to remove from internal object list
di-220	No such object

di-221	NULL data for attribute
di-222	Knots are not non-decreasing
di-223	Weight of a rational control point is less than or equal to zero
di-224	Invalid curve/trimming curve range
di-225	Invalid color spline data
di-226	Invalid address for surface
di-227	Invalid trimming curve approximation type
di-228	Invalid trimming loop
di-229	Invalid hint for nurbs surface
di-230	Invalid data spline data
di-231	NULL Gcache
di-238	Operator not defined on object
di-239	Pcache not associated with any device
di-240	Maximum value for the width or height of a memory raster is 4096
di-241	NULL object
di-242	Invalid error number
di-243	Unsafe Overlay mode detected, this may result in incorrect rendering

GX Frame Buffer Errors

gx-1	Must upgrade GX to FBC rev 1 or later
gx-2	Out of range or invalid floating point data loaded into TEC
gx-3	Can't get GX file descriptor
gx-4	Can't mmap Gx registers
gx-5	Can't mmap Gx frame buffer

gx-6	Selected invalid frame buffer number; selected buffer unchanged
gx-7	Requested zero buffers; number of buffers unchanged
gx-8	Can't unmap Gx registers
gx-9	Can't unmap Gx frame buffer
gx-10	Invalid transform type (integer) for 3d context
gx-11	Improperly dispatched Gx primitive
gx-12	Tried to switch buffers on single buffered GX

Xlib/PEXlib Errors

xpex -1	Unable to create all colors in the PEX color table

A Glossary of XGL Terms

Accumulation Buffer

A buffer containing an array of pixels where the contents of the accumulation process is stored. The accumulation process takes a series of images and combines them. The most common reason for doing multiple rendering and accumulation is to reduce the aliasing effect. An accumulation buffer can also be used to render motion blur, depth of field, or soft shadows.

ASAP

As Soon As Possible. One of two deferral modes that control whether or not the data sent to the device is buffered. If the mode is ASAP, then data will be sent to the device immediately.

ASTI

At **S**ome **TI**me. One of two deferral modes that control whether or not the data sent to the device is buffered. If the mode is ASTI, then data will be sent to the device when the buffer is full; the application does not know when this will happen. It must use `xgl_context_post` to explicitly empty the buffer. The size of the buffer is device-dependent.

Ambient Light

A light source, containing only color, that adds non-directional lighting to a scene. Ambient light adds a uniform illumination to all surfaces, regardless of their orientations.

Attribute

State information associated with an XGL object.

Backing Store

A *backing store* maintains the window contents when a window is unmapped or covered by other windows so that the window is automatically refreshed with the current contents when it becomes visible again.

B-Spline Curve

The type of curve used by XGL for its curve rendering. B-spline curves can represent simple geometric shapes and complex free-form curves.

Class

An abstract data type that defines a virtual graphics resource within XGL. Each class is composed of a set of *operators* (functions) and *attributes* (state information) that specifies its functionality. Applications create instances of classes, where an instance of a class is called an *object*.

CMAP

An acronym for Color Map object.

Color Map Object

Object used to add color to an XGL application and to allow color value conversions from one color type to another.

Color Model

A method of representing the color space of a graphics system. In XGL, colors can be described in one of three color models: RGB, grayscale, or indexed color.

Color Table

An array of RGB color values used to map color indices (indexed color values) to RGB colors.

Concatenation

A method of joining two Transform objects together to produce a single resultant Transform object. Transforms can either be *preconcatenated* or *postconcatenate*d, depending on the order in which they are appended or prepended to the original transform.

Context Object

An XGL object that is an abstraction of a renderer. It contains graphics rendering state information, graphics primitives, and non-primitive operators used for several utility operations (such as copying pixels or clearing a Device).

Context Operator

One or more functions that affect the state of a Context object.

DC

The *Device Coordinate* System. See *Device Coordinates*.

Depth Cueing

An effect that modulates colors of primitives according to depth, enabling the viewer to distinguish whether a primitive is close or far away.

Device Coordinates

A device-dependent coordinate system in which position is usually specified by *x,y* pixel location and, for 3D applications, *z* is specified as a depth in a Z-buffer. The point (0,0,0) is the upper left corner, where the axes are oriented right for positive *x*, down for positive *y*, and away for positive *z*.

Device Object

An object that is an abstraction of a graphics display device. The Device object has two subclasses: *Raster*, which represents a two-dimensional rectangular array of discrete image samples (pixels), and *Non-Raster*, which represents any picture representation, such as CGM, that doesn't rely on discrete pixels to display the image.

DGA

Direct Graphics Access. DGA arbitrates access to the display between XGL and the X11 server. This allows XGL to talk directly to the frame buffer, which results in maximum performance.

Directional Light

A light source that is assumed to be at a nearly infinite distance from the scene so that it emits parallel light rays from a specified direction (similar to light from the sun).

Dithering

Dithering mixes two or more colored pixels to approximate a given color on the screen. This means placing pixels of different colors next to each other so that from a distance, the colors blend together to make a different color. In color dithering, the dither cell is filled with one or more colors defined in the current Color Map. When viewed together, the colors approximate the desired color.

Gcache Object

An object that is used to store an XGL primitive. A GCache can also reduce the complexity of application geometry by allowing the XGL library to process the geometry into simpler forms.

GKS

A specification for a standard 2D graphics library.

Graphics Primitive

A basic graphic element used to draw something on the device (for example, polyline, polygon, text, marker, and so on).

Handle

An opaque pointer to an XGL object provided to the application when an object is created. The application uses it to modify, inquire, or interact with the object.

HLHSR

Hidden **L**ine **H**idden **S**urface **R**emoval. A 3D Context has a rendering pipeline that consists of a lighting and shading stage, a depth cueing stage, and an HLHSR stage. (The HLHSR state may not be the last stage in the pipeline.) The HLHSR stage causes objects that are in front of objects in the 3D scene to obscure objects behind them.

Immediate Mode

A rendering mode in which all drawing functions are sent to the device immediately or sent to a device buffer immediately. There is *no* intermediate storage of drawing commands as there is in a display list.

Indexed Color

Color model in which colors are specified as indices into an array of color values. The array, usually called a color table, can contain color values from any color space (for example, RGB).

Infinite Light

A light source that is infinitely far from the surface that is being illuminated. Infinite light sources produce parallel light rays that are characterized by a direction only.

Instantiation

Allocation of resources that occurs when a variable of an object type (*Xgl_object*) is declared.

Intensity

The brightness of a color value.

Light Object

Object that defines a particular light source that can be used within a 3D Context. Four different types of lights are available: ambient, directional, positional, and spot.

Line Pattern Object

An object that defines line style patterns, used when rendering vectors, curves, and edges.

Marker Object

An object that defines markers.

Marker Primitive

An image that is drawn at a particular point in space.

MC

The *Modeling Coordinate* System. See Modeling Coordinates.

Memory Raster

A Raster object that designates a rectangular area of non-screen memory.

Model Transform

The composite of the Local Model Transform with the Global Model Transform. It is used in mapping geometric *primitives* defined in *Modeling Coordinate (MC)* space to *World Coordinate (WC)* space.

Modeling Coordinates

The coordinate space where a geometric model is defined. Each Modeling Coordinate System is transformed into the application's common World Coordinate space by the Model Transform.

NOP

A shorthand expression, meaning "NO-OP," or non-operator. Depending on a certain state of a machine or software, previously specified instructions are disabled.

NURBS

Non-Uniform Rational B-Spline curve.

Object

An instance of an XGL class that is created by calling an XGL creation operator specific to a particular class. An object is an abstraction of a graphics resource. The object contains various operators and attributes that can be used to manipulate XGL's graphics resources and graphics state.

Operator

Function that controls the behavior of an object.

PHIGS

The Programmer's Hierarchical Interactive Graphics Standard, a specification for a standard 3D graphics library.

PHIGS PLUS

A proposed extension to the PHIGS standard, which includes lighting and shading functionality.

Pick Aperture

The area (2D) or volume (3D) used to test primitives for picking.

Pick ID

The value used to identify a primitive that has been picked.

Picking

The selection of a particular graphics primitive by checking whether it falls within a predefined pick aperture.

Pipeline

A kind of graphics mapping. For example, in XGL, the series of mappings from one coordinate space to another is known as the *transformation pipeline*.

Positional Light

A single light source, specified by a color and a location, which radiates light rays uniformly in all directions (like a bare incandescent light bulb).

Primitive

See *Graphics Primitive.*

Raster

A two-dimensional rectangular area on which images are drawn, used as the output device for all XGL operations.

Rendering

The process of converting geometric data and its attributes into an image on the display.

Rendering Attribute

A description property belonging to graphics primitives (for example, color, width, pattern definition, and so on).

RGB

Color model in which colors are specified as intensities (between 0.0 and 1.0) of the three CRT monitor primary colors, red, green, and blue.

ROP

A raster operation. A Raster OPeration is a function of three variables: a source pixel, a destination pixel, and a per-plane pixel mask. Graphics primitives are rendered using the ROP to combine source and destination.

Rotation

The pivoting of a graphics image about a coordinate axis that is accomplished with a Transform object.

Scaling

The reduction or enlargement of a graphics image that is accomplished with a Transform object.

Spot Light

A light source specified by color, location, and direction, which radiates light rays from a single point with a maximum intensity along the specified direction.

Stroke Font Object

An object that defines the stroke font used by the Context object.

System State Object

An object that maintains state information about all operations occurring during a single XGL session.

Text Local Coordinates

Stroke text is defined in the Text Local Coordinate (TLC) system. A unit vector in TLC is the same length as a unit vector in Model Coordinates. The parameters of the stroke text operators define the orientation of TLC with respect to Modeling Coordinates.

TLC

See *Text Local Coordinates.*

Transform Object

An XGL object that specifies geometric transformations on output primitives The default for all transforms is the identity transform, which uses the identity matrix.

Transformation Matrix

A matrix that specifies a linear mapping of one coordinate space to another coordinate space.

Transformation Pipeline

The series of transformations used in mapping geometric data and their attributes from Model Coordinate (MC) space to Device Coordinate (DC) space.

Translation

Movement of a graphics image relative to the origin of the coordinate system. Accomplished with a Transform object.

VDC

The *Virtual Device Coordinate* system. See *Virtual Device Coordinates.*

VDC Transform

The final transform in the transformation pipeline that provides the mapping between Virtual Device Coordinates (VDC) and Device Coordinates (DC).

VDC Transformation Matrix

The transformation matrix used to map the clip-space window onto a rectangular region of the Raster known as the *Raster viewport.*

View Model

A model that specifies the geometric aspects of image formation, determining the orientation of images and the spatial relationships between objects.

View Transform

The transform used in mapping graphic objects defined in World Coordinate (WC) space to Normalized Device Coordinate (NDC) space where viewing operations take place.

View Transformation Matrix

The transformation matrix used in mapping graphic objects from WC to NDC. The view transformation matrix can be specified directly or derived from the matrix product of the view orientation matrix and the view mapping matrix.

Virtual Device Coordinates

A device-independent coordinate system used to isolate the specification of view boundaries from device coordinates.

WC

The *World Coordinate System.* See *World Coordinates.*

Window Raster

A *Raster object* that designates a rectangular area on the screen of the display device.

World Coordinates

The application's common coordinate system. If a Model Transform exists, then it transforms the Model Coordinate data to World Coordinates. If one does not exist, the application's data will be initially defined in World Coordinate space. Individual objects in the image to be rendered are represented in World Coordinates in proper perspective (3D only) with respect to each other.

Index

A

acceleration
  inquiring hardware acceleration, 86
  with DGA display technology, 2
  XGL and hardware graphics acceleration, 1
accumulation buffer, 222
  accumulation buffer depth, 225
  antialiasing of images, 222
  destination buffer for accumulation, 223
  jitter offset, 225
  xgl_context_accumulate, 223
annotation primitives, 184
annotation text, 318
antialiasing
  accumulation buffer, 222, 567
  edges, 192
  lines, 189
  markers, 190, 377
  stroke text, 198
  surfaces, 193

B

background color, 186
backing store, 63
bitmap fonts, 339
bounding boxes, 179
  bounding box data structures, 181
  geometry status of bounding boxes, 182
  xgl_context_check_bbox, 182
B-spline curves
  approximation type, 235
    chordal deviation, 235
    metric, 235
    relative, 236
    static, 235
  approximation value, 235
  control points, 231
    homogeneous, 231
  dynamic tessellation, 235
  knot sequence, 232
  mathematical definition, 230
  non-uniform knot vector, 232
  parameter range, 232, 234
  rational curves, 232
  rendering a NURBS curve, 233
  static tessellation, 235
  uniform knot vector, 232
  with color, 237
  xgl_nurbs_curve, 233
B-spline surfaces
  and texture mapping, 478
  approximation criteria, 255
    chordal deviation, 255

metric, 255
relative, 255
static, 255
approximation value, 255
control points, 248
homogeneous, 248
isoparametric curves, 257
knots, 248
mathematical definition, 247
performance hints, 253
rational surfaces, 248
rendering a NURBS surface, 249
silhouette edges, 258, 259
surface trimming, 250
curve orientation rule, 251
odd-winding rule, 251
trimming curves, 250
trimming loops, 250
trimming curves
approximation criteria, 256
surface boundaries, 257
with color, 258
xgl_nurbs_surface, 249
buffering primitive data, 164
buffers
accumulation buffer, 222
copying buffer contents
xgl_context_copy_buffer, 213
xgl_image, 214
Z-buffer, 188

C

caching geometric data, 429
CGM Metafile object, 66
closing a CGM file, 68
creating a CGM device, 66
creating a CGM file, 68
encoding formats, 69
line patterns, 71
markers, 72
XGL_CGM_DESCRIPTION, 69
XGL_CGM_ENCODING, 69
XGL_CGM_PICTURE_DESCRIPTION, 70
XGL_CGM_SCALE_FACTOR, 70
XGL_CGM_SCALE_MODE, 70
XGL_CGM_TYPE, 69
XGL_CGM_VDC_EXTENT, 71
XGL_DEV_COLOR_MAP, 71
channel information, 453
clearing the DC viewport, 162
clipping, 292
markers, 374
model clipping in 3D, 303
optimizing with bounding boxes, 179
view clipping planes, 299
color
and lighting, 397
indexed lighting, 404
RGB lighting, 403
color shading, 400
reflected color, 398
color conversion, 117
color cube, 117
color map double buffering, 144
Color Map object, 118
and Device object, 119
color cube, 130
color map double buffering, 144
color ramps, 124
color table, 119
creating a color map object, 118
creating a color table, 120
creating color ramps, 125
default color table, 119
dithering, 131
in CGM Metafile, 71
mapping colors between color types, 130
mapping X pixels, 133
plane masks and double buffering, 145
XGL_CMAP_COLOR_CUBE_SIZE, 131
XGL_CMAP_COLOR_MAPPER, 130
XGL_CMAP_COLOR_TABLE, 119, 500

XGL_CMAP_COLOR_TABLE_SIZE, 119
XGL_CMAP_DITHER_MASK, 132
XGL_CMAP_INVERSE_COLOR_MAPPER, 130
XGL_CMAP_NAME, 132
XGL_CMAP_RAMP_LIST, 125
XGL_CMAP_RAMP_NUM, 125
XGL_DEV_COLOR_MAP, 58
XGL_DEV_COLOR_TYPE, 58
color mapping, 130
compiling
XGL application, 15
XGL example programs, 18
Context object, 159
associating with other objects, 160
creating a Context object, 160
drawing primitives, summary, 173
environment state, 160
example programs, 204, 215
graphics state, 160
hidden line and surface removal, 188
non-drawing primitives, summary, 162
rendering pipeline, 397
setting the paint type for overlay windows, 98
texture mapping attributes, 467, 477
transforms, 293
turning lights on and off, 407
XGL_3D_CTX_BLEND_DRAW_MODE, 195
XGL_3D_CTX_BLEND_FREEZE_ZBUFFER, 195
XGL_3D_CTX_DEPTH_CUE_COLOR, 197, 414
XGL_3D_CTX_DEPTH_CUE_INTERP, 197
XGL_3D_CTX_DEPTH_CUE_MODE, 197, 413
XGL_3D_CTX_DEPTH_CUE_REF_PLANES, 197, 414
XGL_3D_CTX_DEPTH_CUE_SCALE_FACTORS, 197
XGL_3D_CTX_DEPTH_CURE_SCALE_FACTORS, 414
XGL_3D_CTX_EDGE_Z_OFFSET, 192
XGL_3D_CTX_HLHSR_DATA, 188
XGL_3D_CTX_HLHSR_MODE, 188
XGL_3D_CTX_LIGHT_NUM, 167
XGL_3D_CTX_LIGHT_SWITCHES, 194
XGL_3D_CTX_LIGHTS, 168
XGL_3D_CTX_LINE_COLOR_INTERP, 190
XGL_3D_CTX_NORMAL_TRANS, 298
XGL_3D_CTX_SURF_BACK_AMBIENT, 194
XGL_3D_CTX_SURF_BACK_COLOR, 193
XGL_3D_CTX_SURF_BACK_COLOR_SELECTOR, 193
XGL_3D_CTX_SURF_BACK_DIFFUSE, 194
XGL_3D_CTX_SURF_BACK_FILL_STYLE, 193
XGL_3D_CTX_SURF_BACK_FPAT, 193
XGL_3D_CTX_SURF_BACK_FPAT_POSITION, 193
XGL_3D_CTX_SURF_BACK_ILLUMINATION, 194
XGL_3D_CTX_SURF_BACK_SPECULAR, 194
XGL_3D_CTX_SURF_BACK_TRANSP, 195
XGL_3D_CTX_SURF_DC_OFFSET, 195
XGL_3D_CTX_SURF_FACE_CULL, 194
XGL_3D_CTX_SURF_FACE_DISTINGUISH, 193
XGL_3D_CTX_SURF_FRONT_AMBIENT, 194
XGL_3D_CTX_SURF_FRONT_DIFFUSE, 194
XGL_3D_CTX_SURF_FRONT_DMAP, 196, 197, 477

XGL_3D_CTX_SURF_FRONT_DMAP_NUM, 196, 197, 477
XGL_3D_CTX_SURF_FRONT_DMAP_SWITCHES, 196, 197, 477
XGL_3D_CTX_SURF_FRONT_ILLUMINATION, 194
XGL_3D_CTX_SURF_FRONT_SPECULAR, 194
XGL_3D_CTX_SURF_FRONT_TMAP, 467
XGL_3D_CTX_SURF_FRONT_TMAP_NUM, 468
XGL_3D_CTX_SURF_FRONT_TMAP_SWITCHES, 468
XGL_3D_CTX_SURF_FRONT_TRANSP, 195
XGL_3D_CTX_SURF_GEOM_NORMAL, 194
XGL_3D_CTX_SURF_NORMAL_FLIP, 194
XGL_3D_CTX_SURF_SILHOUETTE_EDGE_FLAG, 195
XGL_3D_CTX_SURF_TRANSP_BLEND_EQ, 195
XGL_3D_CTX_SURF_TRANSP_METHOD, 195
XGL_3D_CTX_Z_BUFFER_COMP_METHOD, 188
XGL_3D_CTX_Z_BUFFER_WRITE_MASK, 188
XGL_3D_DEPTH_CUE_INTERP, 414
XGL_CTX_AA_FILTER_SHAPE, 189
XGL_CTX_ARC_FILL_STYLE, 193
XGL_CTX_ATEXT_ALIGN_VERT, 199
XGL_CTX_ATEXT_CHAR_HEIGHT, 199
XGL_CTX_ATEXT_CHAR_SLANT_ANGLE, 199
XGL_CTX_ATEXT_CHAR_UP_VECTOR, 199
XGL_CTX_ATEXT_HORIZ, 199
XGL_CTX_ATEXT_PATH, 199
XGL_CTX_ATEXT_STYLE, 199
XGL_CTX_BACKGROUND_COLOR, 186
XGL_CTX_CLIP_PLANES, 200
XGL_CTX_CURVE_APPROX_VAL, 235
XGL_CTX_DEFERRAL_MODE, 164
XGL_CTX_DEVICE, 165
XGL_CTX_EDGE_AA_BLEND_EQ, 192
XGL_CTX_EDGE_AA_FILTER_SHAPE, 192
XGL_CTX_EDGE_AA_FILTER_WIDTH, 192
XGL_CTX_EDGE_ALT_COLOR, 192
XGL_CTX_EDGE_COLOR, 192
XGL_CTX_EDGE_PATTERN, 192
XGL_CTX_EDGE_STYLE, 192
XGL_CTX_EDGE_WIDTH_SCALE_FACTOR, 192
XGL_CTX_FRONT_COLOR, 193
XGL_CTX_GLOBAL_MODEL_TRANS, 200, 297
XGL_CTX_LINE_AA_BLEND_EQ, 189
XGL_CTX_LINE_AA_FILTER_WIDTH, 189
XGL_CTX_LINE_ALT_COLOR, 189
XGL_CTX_LINE_CAP, 189
XGL_CTX_LINE_COLOR, 189
XGL_CTX_LINE_COLOR_SELECTOR, 189
XGL_CTX_LINE_JOIN, 189
XGL_CTX_LINE_MITER_LIMIT, 189
XGL_CTX_LINE_PATTERN, 189
XGL_CTX_LINE_STYLE, 189
XGL_CTX_LINE_WIDTH_SCALE_FACTOR, 190
XGL_CTX_LOCAL_MODEL_TRANS, 200, 297
XGL_CTX_MARKER, 190
XGL_CTX_MARKER_AA_BLEND_EQ, 190
XGL_CTX_MARKER_AA_FILTER_SHAPE, 190
XGL_CTX_MARKER_AA_FILTER_WIDTH, 190

XGL_CTX_MARKER_COLOR, 190
XGL_CTX_MARKER_COLOR_SELECTOR, 190
XGL_CTX_MARKER_SCALE_FACTOR, 190
XGL_CTX_MAX_TESSELLATION, 191
XGL_CTX_MC_TO_DC_TRANS, 302
XGL_CTX_MIN_TESSELLATION, 191
XGL_CTX_MODEL_TRANS, 200, 297
XGL_CTX_MODEL_TRANS_STACK_SIZE, 164
XGL_CTX_NEW_FRAME_ACTION, 186
XGL_CTX_NURBS_CURVE_APPROX, 191, 235
XGL_CTX_NURBS_CURVE_APPROX_VAL, 191
XGL_CTX_NURBS_SURF_APPROX, 196, 254
XGL_CTX_NURBS_SURF_PARAM_STYLE, 196, 257
XGL_CTX_PICK_APERTURE, 166
XGL_CTX_PICK_BUFFER_SIZE, 166
XGL_CTX_PICK_ENABLE, 166
XGL_CTX_PICK_STYLE, 166
XGL_CTX_PLANE_MASK, 145, 186
XGL_CTX_RASTER_FILL_STYLE, 186
XGL_CTX_RASTER_FPAT, 186
XGL_CTX_RASTER_FPAT_POSITION, 186
XGL_CTX_RASTER_STIPPLE_COLOR, 186
XGL_CTX_RENDER_BUFFER, 165, 187
XGL_CTX_RENDERING_ORDER, 165
XGL_CTX_ROP, 186
XGL_CTX_SFONT_n, 198
XGL_CTX_STEXT_AA_BLEND_EQ, 198
XGL_CTX_STEXT_AA_FILTER_SHAPE, 198
XGL_CTX_STEXT_AA_FILTER_WIDTH, 198
XGL_CTX_STEXT_ALIGHT_HORIZ, 198
XGL_CTX_STEXT_ALIGN_VERT, 198
XGL_CTX_STEXT_CHAR_ENCODING, 198
XGL_CTX_STEXT_CHAR_EXPANSION_FACTOR, 198
XGL_CTX_STEXT_CHAR_HEIGHT, 198
XGL_CTX_STEXT_CHAR_SLANT_ANGLE, 198
XGL_CTX_STEXT_CHAR_SPACING, 198
XGL_CTX_STEXT_CHAR_UP_VECTOR, 198
XGL_CTX_STEXT_COLOR, 198
XGL_CTX_STEXT_PATH, 198
XGL_CTX_STEXT_PRECISION, 198
XGL_CTX_SURF_AA_BLEND_EQ, 193
XGL_CTX_SURF_AA_FILTER_SHAPE, 193
XGL_CTX_SURF_AA_FILTER_WIDTH, 193
XGL_CTX_SURF_EDGE_FLAG, 192
XGL_CTX_SURF_FRONT_COLOR_SELECTOR, 193
XGL_CTX_SURF_FRONT_FILL_STYLE, 193
XGL_CTX_SURF_FRONT_FPAT, 193
XGL_CTX_SURF_FRONT_FPAT_POSITION, 193
XGL_CTX_SURF_INTERIOR_RULE, 194
XGL_CTX_THRESHOLD, 186
XGL_CTX_VDC_MAP, 200
XGL_CTX_VDC_ORIENTATION, 167
XGL_CTX_VIEW_TRANS, 200

coordinate systems, 292
- device coordinates, 293
- modeling coordinates, 292

virtual device coordinates, 293
world coordinates, 292
copying buffer contents, 163

D

Data Map Texture object
example program, 482
texture descriptors
channel information, 475
describing the MipMap, 476
filters, 466, 475
texture sampling, 466, 475
texture type, 475
wrapping the texture, 476
XGL_DMAP_TEXTURE_DESCRIPTORS, 474
XGL_DMAP_TEXTURE_NUM_DESCRIPTORS, 474
XGL_DMAP_TEXTURE_ORIENTATION_MATRIX, 477
XGL_DMAP_TEXTURE_PARAM_TYPE, 478
XGL_DMAP_TEXTURE_U_INDEX, 478, 479
XGL_DMAP_TEXTURE_V_INDEX, 478, 479
data structures
bounding boxes, 179
facet structures, 178
point lists, 177
point types, 177
specialized structures, 178
DC (device coordinates), 293
defaults
default objects at XGL initialization, 36
light type, 407
depth cueing, 411
applications of, 411
attributes, 413
linear, 412
rendering pipeline, 396
scaled, 412
Device object, 53
CGM device, 66
creating a CGM device object, 66
creating a Device object, 54
example programs, 74, 88
Memory Raster, 54
Stream device, 57
window description parameter, 55
Window Raster, 54
XGL_DEV_COLOR_MAP, 58
XGL_DEV_COLOR_TYPE, 58
XGL_DEV_CONTEXT, 60
XGL_DEV_CONTEXTS_NUM, 59
XGL_DEV_MAXIMUM_COORDINATES, 60
XGL_DEV_REAL_COLOR_TYPE, 58
XGL_MEM_RAS_IMAGE_BUFFER_ADDR, 64
XGL_MEM_RAS_Z_BUFFER_ADDR, 65
XGL_RAS_DEPTH, 60
XGL_RAS_HEIGHT, 60
XGL_RAS_RECT_LIST, 61
XGL_RAS_RECT_NUM, 61
XGL_RAS_SOURCE_BUFFER, 61
XGL_RAS_WIDTH, 60
XGL_WIN_RAS_BACKING_STORE, 40, 63
XGL_WIN_RAS_BUF_DISPLAY, 62
XGL_WIN_RAS_BUF_DRAW, 62
XGL_WIN_RAS_BUF_MIN_DELAY, 62
XGL_WIN_RAS_BUFFERS_ALLOCATED, 63
XGL_WIN_RAS_BUFFERS_SUPPORTED, 62
XGL_WIN_RAS_DESCRIPTOR, 61
XGL_WIN_RAS_PIXEL_MAPPING, 61, 133
XGL_WIN_RAS_POSITION, 61
XGL_WIN_RAS_STEREO_MODE, 62
XGL_WIN_TYPE, 61
DGA (Direct Graphics Access)
display technology, 2, 4

local rendering, 21
mixing XGL and Xlib calls, 73
DGA transparent overlay, 95
DirectColor, 59
dithering, 131
double buffering
hardware, 62

E

edges
edge color, 364
edge pattern, 362
edge rendering attributes,
summary, 192
environment texture mapping, 470
environment variable
OPENWINHOME, 15
XGLHOME, 15, 45
error reporting
notification function, 46
system errors, 48
example programs
Bezier curves, 237
circle primitive, 209
color cubes, 134
color map double buffering, 145
color ramps, 126
copying pixels, 217
facet shading, 415
Gcache with complex polygon, 447
Gcache with simple polygon, 444
getting and setting pixel values, 215
inquire, 88
linear depth cueing, 422
NURBS circles, 242
NURBS surface, 259
OLIT and XGL, 83
patterned lines, 364
patterned polygon edges, 368
perspective viewing, 305
picking, 389
polygon primitive, 204
predefined markers, 377
raster text, 342
rectangle primitive, 206
rotating geometry, 282
scaled depth cueing, 425
scaling geometry, 284
simple color example, 121
simple XGL program, 30
stroke text, 334
texture mapping, 482
transforming 3D geometry, 286
translating geometry, 280
transparent overlay windows, 99
user-defined markers, 380
vertex shading, 420
Xlib and XGL, 75
XView and XGL, 79

F

facets
color, 178
determining front or back facing, 178
facet normals, 178
facet shading, 400
surface primitives, 397
filled areas, 170
font files, location of, 45
font files, location of stroke fonts, 314
fonts
bitmap, 339
stroke, 314

G

Gcache object
available primitives, 437
creating a Gcache, 437
displaying cached geometry, 443
example programs, 444
when to use, 436
XGL_GCACHE_BYPASS_CLIP, 440
XGL_GCACHE_DISPLAY_PRIM_
TYPE, 440
XGL_GCACHE_DO_POLYGON_
DECOMP, 440

XGL_GCACHE_FACET_LIST_LIST, 441
XGL_GCACHE_IS_EMPTY, 441
xgl_gcache_multi_elliptical_arc, 437
xgl_gcache_multi_simple_polygon, 437
xgl_gcache_multimarker, 437
xgl_gcache_multipolyline, 438
xgl_gcache_nurbs_curve, 438
XGL_GCACHE_NURBS_CURVE_MODE, 441
XGL_GCACHE_NURBS_SURF_MODE, 442
xgl_gcache_nurbs_surface, 438
XGL_GCACHE_ORIG_PRIM_TYPE, 440
xgl_gcache_polygon, 439
XGL_GCACHE_POLYGON_TYPE, 440
XGL_GCACHE_PT_LIST_LIST, 440
XGL_GCACHE_SHOW_DECOMP_EDGES, 440
xgl_gcache_stroke_text, 439
xgl_gcache_triangle_list, 439
xgl_gcache_triangle_strip, 439
XGL_GCACHE_USE_APPL_GEOM, 441

getting pixel values, 213
glossary, 567
graphics resource, 27

H

hidden lines, 188
hidden surfaces, 188
HLHSR (hidden line and surface removal), 188
 and transparent surfaces, 195
 mode, 188
 summary of attributes, 188
 using surface edges, 188
 Z-buffering, 188

I

illumination types, 400
immediate mode graphics libraries, 4
immediate mode graphics library, 1
immediate-mode rendering, 2
indexed color, 116
 and lighting, 404
inquiring XGL acceleration features, 86

L

Light object, 407
 copying a Light object, 409
 creating a Light object, 407
 defining a light type, 409
 example programs, 415
 XGL_LIGHT_ATTENUATION_n, 409
 XGL_LIGHT_COLOR, 409
 XGL_LIGHT_DIRECTION, 409
 XGL_LIGHT_POSITION, 409
 XGL_LIGHT_SPOT_ANGLE, 409
 XGL_LIGHT_SPOT_EXPONENT, 409
 XGL_LIGHT_TYPE, 409
light sources, 399
 ambient light, 399
  reflectance components, 402
 color, 397
 directional light, 399
  reflectance components, 402
 positional light, 399
  reflectance components, 402
 spot light, 399
  reflectance components, 403
lighting
 color, 397
 copying a Light object, 409
 creating a Light object, 407
 default light type, 407
 equations, 400
 indexed lighting, 404
 light attenuation equation, 403
 light sources, 399

lighting variables, 401
reflectance components, 402
reflected color, 398
reflection components, 398
reflection types, 398
rendering pipeline, 395
RGB lighting, 403
turning lights on and off, 407
Line Pattern object, 355
creating a Line Pattern object, 357
defining a line pattern, 358
edge color, 364
edge patterns, 362
example programs, 364
line pattern color, 362
line pattern segments, 358
predefined line patterns, 356
XGL_CTX_LINE_ALT_COLOR, 362
XGL_CTX_LINE_COLOR, 362
XGL_LPAT_DATA, 359
XGL_LPAT_DATA_SIZE, 359
XGL_LPAT_OFFSET, 360
lines, 170
alternate color for patterned lines, 189
antialiasing, 189
corner mitering, 189
defining a line pattern, 358
join styles, 189
line cap, 189
line color, 189, 362
line pattern, 189
line rendering attributes, summary, 189
line style, 189, 361
line width scale factor, 190
predefined line patterns, 356
loadable device pipeline architecture, 2

M

Marker object, 373
creating a Marker object, 375
example programs, 377
predefined markers, 374
XGL_CTX_MARKER, 376
XGL_CTX_MARKER_COLOR, 376
XGL_CTX_MARKER_SCALE_FACTOR, 376
xgl_multimarker, 376
markers, 170, 373
antialiasing, 190, 377
marker color, 190
marker descriptions, 375
marker rendering attributes, summary, 190
marker scale factor, 190
predefined markers, 374
material color, 397
MC (modeling coordinates), 292
Memory Raster object
caching bitmap fonts, 340
in texture mapping, 453
XGL_DEV_COLOR_MAP, 58
XGL_MEM_RAS_IMAGE_BUFFER_ADDR, 64
XGL_MEM_RAS_Z_BUFFER_ADDR, 65
XGL_RAS_DEPTH, 60
XGL_RAS_HEIGHT, 60
XGL_RAS_WIDTH, 60
MIP map data, 452
MipMap Texture object
boundary values, 455
creating a MipMap Texture object, 452
determining MIP map size, 458
generating a MIP map, 455
MIP map descriptors, 464, 476
xgl_mipmap_texture_build, 455
XGL_MIPMAP_TEXTURE_LEVELS, 455
XGL_MIPMAP_TEXTURE_MEM_RAS_LIST, 457
modeling coordinates
global modeling transform, 297
local modeling transform, 297
model transform, 297

N

non-uniform rational B-spline curves, 230
NURBS surfaces, *See* B-spline surfaces

O

object instantiation, 27
objects
    overview of XGL objects, 6 to 14
    Color Map, 118
    Context, 159
    Device, 53
        CGM device object, 54
        Memory Raster object, 53
        Stream object, 54
        Window Raster object, 53
    Gcache, 436
    Light, 407
    Line Pattern, 355
    Marker, 373
    MipMap Texture, 452
    object relationships, 29
    Pcache, 430
    Stroke Font, 314
    System State, 43
    Transform, 267
OpenWindows, 73
overlay window, 94

P

Pcache object
    available attributes, 433
    available primitives, 432
    creating a Pcache object, 432
    displaying cached geometry, 434
    performance, 432
    when to use, 431
    XGL_PCACHE_CONTEXT, 432
    xgl_pcache_display, 434
perspective transformation, 299
PEX, 19
picking, 385
    clearing the pick buffer, 388
    identifying the picked primitives, 388
    pick aperture, 385
    pick buffer, 385
    pick buffer size, 166
    pick event, 385
    pick identifier, 385
    polygon picking, 166
    storing pick events, 166
    XGL_CTX_PICK_APERTURE, 385
    XGL_CTX_PICK_BUFFER_SIZE, 386
    XGL_CTX_PICK_ENABLE, 386
    XGL_CTX_PICK_ID_1 _2, 387
    XGL_CTX_PICK_STYLE, 387
    XGL_CTX_PICK_SURF_STYLE, 385
    xgl_pick_clear, 388
    xgl_pick_get_identifiers, 388
pixel operations, 163, 213
pixel plane mask, 186
point lists, 177
points, 170
posting pending graphics primitives, 162
primitives
    overview, 169 to 184
    bounding boxes, 179
    facet structures, 178
    line, 170
    marker, 170
    point, 170
    raster level primitives, 163, 213
    rendering parallel to the projection plane, 184
    surface, 170
    text, 171
PseudoColor, 59

R

raster text
    caching font information, 340
    overview, 339
    rendering, 341
    stencils, 339
reflection
    color, 398
    reflectance components, 402

reflection components, 398
reflection types
ambient reflection, 398
diffuse reflection, 398
specular reflection, 398
rendering pipeline
color, 397
depth cueing, 396
indexed color, 404
lighting, 395
RGB lighting, 403
shading, 396
RGB color, 116
and lighting, 403

S

saving attributes onto a stack, 163
setting pixel values, 213
shading
facet shading, 400
methods, 400
rendering pipeline, 396
shading techniques, 399
vertex shading, 400
stencil text, 339
string encoding
ISO characters, 331
multi-byte, 332
Stroke Font object
character angle, 323
character height, 321
character spacing, 322
character up vector, 323
character width, 322
creating a Stroke Font object, 315
location of stroke font files, 314
specifying encoding schemes, 328
supported fonts, 315
text color, 327
text coordinate system, 316
text path, 324
text precision, 327
xgl_annotation_text, 321
XGL_CTX_ATEXT_CHAR_SLANT_ANGLE, 323
XGL_CTX_ATEXT_STYLE, 329
XGL_CTX_SFONT_x, 321
XGL_CTX_STEXT_ALIGN_HORIZ, 325
XGL_CTX_STEXT_ALIGN_VERT, 325
XGL_CTX_STEXT_CHAR_ENCODING, 321, 328
XGL_CTX_STEXT_CHAR_EXPANSION_FACTOR, 322
XGL_CTX_STEXT_CHAR_HEIGHT, 321
XGL_CTX_STEXT_CHAR_SLANT_ANGLE, 323
XGL_CTX_STEXT_CHAR_SPACING, 322
XGL_CTX_STEXT_CHAR_UP_VECTOR, 323
XGL_CTX_STEXT_COLOR, 327
XGL_CTX_STEXT_PATH, 324
XGL_CTX_STEXT_PRECISION, 327
XGL_SFONT_COMMENT, 320
XGL_SFONT_DEFAULT_CHARACTER, 320
xgl_stroke_text, 321
xgl_stroke_text_extent, 321
surfaces
attributes summary, 191
depth cueing attribute summary, 197
fill attribute summary, 193
lighting attribute summary, 193
mapping a texture to a surface, 451
NURBS surface attribute summary, 196
rendering pipeline, 397
See also B-spline surfaces
surface edge color, 364
surface edge patterns, 363
surface primitives, 170
transparency attribute summary, 195
System State object, 43
location of font files, 45
xgl_close, 44

xgl_open, 43
XGL_SYS_ST_ERROR_DETECTION, 44
XGL_SYS_ST_ERROR_NOTIFICATION_FUNCTION, 44, 45
XGL_SYS_ST_FONT_DIRECTORY, 314
XGL_SYS_ST_SFONT_DIRECTORY, 45
XGL_SYS_ST_VERSION, 45

T

text, 171
- annotation text attribute summary, 199
- character angle, 323
- character height, 321
- character spacing, 322
- character up vector, 323
- character width, 322
- color, 327
- determining text extent, 329
- encoding schemes, 328
- ISO encoding, 314
- location of stroke font files, 314
- multi-byte encoding, 314
- raster text, 339
- rendering parallel to the display surface, 318
- Stroke Font object, 314
- stroke text attribute summary, 198
- supported stroke fonts, 315
- text extent, 320
- text path, 324
- text precision, 327

Texture Map object
- color composition, 460
- composition method, 460
- creating a Texture Map object, 459
- describing the MipMap, 464
- MIP map boundary conditions, 464
- MIP map depth sampling, 466
- orientation matrix, 467
- parameterization method, 470
- perspective correction, 468
- rendering component, 461
- specifying texture coordinate source, 469
- texture descriptor, 459
- texture domain, 473
- texture sampling method, 466
- texture type, 460
- XGL_3D_CTX_SURF_TMAP_PERSP_CORRECTION, 468
- XGL_TMAP_COORD_SOURCE, 469
- XGL_TMAP_DOMAIN, 473
- XGL_TMAP_PARAM_TYPE, 470
- XGL_TMAP_T0_INDEX, 470
- XGL_TMAP_T1_INDEX, 470
- XGL_TMAP_TEXTURE_DESCRIPTOR, 459

texture mapping, 449
- blend mode, 460, 462
- boundary clamping, 464
- boundary conditions, 464
- color compositing methods, 462
  - channel information, 463
- creating a MipMap Texture object, 452
- decal mode, 460, 462
- defining texture control parameters, 474
- environment texture mapping, 470
- example program, 482
- mapping the texture to a surface primitive, 479
- MIP map depth sampling, 466
- MIP map levels, 455
- specifying texture properties, 458
- texture objects, 451, 458
- using the texture through a 3D Context, 467, 477
- vertex-level texture mapping, 469
- xgl_mipmap_texture_build, 455

Transform object, 267
- creating a Transform object, 297
- default Transform objects, 296
- example programs, 276
- matrix groups, 272

using the default viewing
transforms, 306
XGL_TRANS_DATA_TYPE, 276
XGL_TRANS_DIMENSION, 276
xgl_transform_copy, 273
xgl_transform_identity, 273
xgl_transform_invert, 274
xgl_transform_multiply, 274
xgl_transform_point, 275
xgl_transform_point_list, 275
xgl_transform_read, 271
xgl_transform_rotate, 269
xgl_transform_scale, 270
xgl_transform_translate, 271
xgl_transform_transpose, 275
xgl_transform_write, 273
xgl_transform_write_specific, 271
transformation matrices, 267
copying, 273
identity transform, 273
inverse of a matrix, 274
multiplying matrices, 274
reading and writing, 271
rotation, 269
scale, 270
transforming a point, 275
transforming a point list, 275
translate, 270
transpose of a matrix, 275
transformation pipeline, 291
coordinate systems, 292
device coordinates, 293
model clipping in 3D, 303
modeling transform, 292
viewing space, 293
world coordinate space, 292
transforms
MC-to-DC transform, 302
modeling transforms, 293
global modeling transform, 297
local modeling transform, 297
normal transform, 298
normal transform, 298
rotation, 269
scale, 270
translation, 270
VDC transform, 299
viewing transform, 293, 298
transparency
blended, 195
screen-door, 195
transparent overlay windows, 94
choosing an overlay visual, 96
color maps, 98
creating an overlay window, 96
rendering to an overlay window, 98
XGL_CTX_NEW_FRAME_PAINT_
TYPE, 99
XGL_CTX_PAINT_TYPE, 98
TrueColor, 59

V

VDC (virtual device coordinates), 293, 299
window resize, 299
vertex shading, 400
vertex-level texture mapping, 469
view clipping, 298, 304
view mapping
of perspective projection, 299
to device coordinates, 299
to virtual device space, 298
to world coordinates, 297
VDC orientation, 301
view model, 291
coordinate systems, 292
transforms, 293
view orientation, 298, 301
viewing attributes, summary, 200
viewing pipeline, 291
viewport, 299
visual type, 59

W

WC (world coordinate system), 292
Window Raster object
creating a Window Raster, 55
XGL_DEV_COLOR_MAP, 58

XGL_RAS_DEPTH, 60
XGL_RAS_HEIGHT, 60
XGL_RAS_WIDTH, 60
XGL_WIN_RAS_BACKING_STORE, 63
XGL_WIN_RAS_BUF_DISPLAY, 62
XGL_WIN_RAS_BUF_DRAW, 62
XGL_WIN_RAS_BUF_MIN_DELAY, 62
XGL_WIN_RAS_BUFFERS_ALLOCATED, 63
XGL_WIN_RAS_BUFFERS_REQUESTED, 62
XGL_WIN_RAS_DESCRIPTOR, 61
XGL_WIN_RAS_PIXEL_MAPPING, 61
XGL_WIN_RAS_POSITION, 61
XGL_WIN_RAS_TYPE, 61

world coordinates, 298

X

X events
- window repaint, 38
- window resize, 74, 299
- xgl_window_raster_resize, 74

X windows
- mixing XGL and Xlib calls, 73
- sharing the X color map, 132
- transparent overlay windows, 94
- visual class and Window Raster color type, 58
- window descriptor for XGL Device, 55
- X11 bitmap fonts, 339
- XGL and the X environment, 1, 20, 73

XGL
- overview, 1
- and PEX, 19
- as object-based system, 6, 27
- basic concepts
  - object-based programming model, 27
  - XGL attributes, 29
  - XGL object relationships, 29
- default objects, 36
- limitations, 40
- local rendering, 22
- opening XGL, 34
- programming tips, 37
- remote rendering, 22
- simple example program, 30
- window resize, 299
- XGL and the X environment, 20

XGL_3D_CTX_ACCUM_OP_DEST, 187
XGL_3D_CTX_BLEND_DRAW_MODE, 195
XGL_3D_CTX_BLEND_FREEZE_ZBUFFER, 195
XGL_3D_CTX_DEPTH_CUE_COLOR, 197
XGL_3D_CTX_DEPTH_CUE_INTERP, 197
XGL_3D_CTX_DEPTH_CUE_MODE, 197
XGL_3D_CTX_DEPTH_CUE_REF_PLANES, 197
XGL_3D_CTX_DEPTH_CUE_SCALE_FACTORS, 197
XGL_3D_CTX_EDGE_Z_OFFSET, 192
XGL_3D_CTX_HLHSR_DATA, 188
XGL_3D_CTX_HLHSR_MODE, 188
XGL_3D_CTX_JITTER_OFFSET, 187, 225
XGL_3D_CTX_LIGHT_NUM, 407, 411
XGL_3D_CTX_LIGHT_SWITCHES, 194, 411
XGL_3D_CTX_LIGHTS, 411
XGL_3D_CTX_LINE_COLOR_INTERP, 190
XGL_3D_CTX_MODEL_CLIP_PLANES, 201, 303
XGL_3D_CTX_MODEL_CLIP_PLANES_NUM, 201
XGL_3D_CTX_NORMAL_TRANS, 200, 298
XGL_3D_CTX_SURF_BACK_AMBIENT, 194, 410
XGL_3D_CTX_SURF_BACK_COLOR, 193, 410

XGL_3D_CTX_SURF_BACK_COLOR_SELECTOR, 193, 410
XGL_3D_CTX_SURF_BACK_DIFFUSE, 194, 410
XGL_3D_CTX_SURF_BACK_FILL_STYLE, 193
XGL_3D_CTX_SURF_BACK_FPAT, 193
XGL_3D_CTX_SURF_BACK_FPAT_POSITION, 193
XGL_3D_CTX_SURF_BACK_ILLUMINATION, 194, 410
XGL_3D_CTX_SURF_BACK_LIGHT_COMPONENT, 194, 410
XGL_3D_CTX_SURF_BACK_SPECULAR, 194, 410
XGL_3D_CTX_SURF_BACK_SPECULAR_COLOR, 194, 410
XGL_3D_CTX_SURF_BACK_SPECULAR_POWER, 194, 410
XGL_3D_CTX_SURF_BACK_TMAP_NUM, 468
XGL_3D_CTX_SURF_BACK_TMAP_SWITCHES, 468
XGL_3D_CTX_SURF_BACK_TRANSP, 195
XGL_3D_CTX_SURF_DC_OFFSET, 195
XGL_3D_CTX_SURF_FACE_CULL, 194, 410
XGL_3D_CTX_SURF_FACE_DISTINGUISH, 193, 410
XGL_3D_CTX_SURF_FRONT_AMBIENT, 194, 410
XGL_3D_CTX_SURF_FRONT_DIFFUSE, 194, 410
XGL_3D_CTX_SURF_FRONT_DMAP, 196, 197, 477
XGL_3D_CTX_SURF_FRONT_DMAP_NUM, 196, 197, 477
XGL_3D_CTX_SURF_FRONT_DMAP_SWITCHES, 196, 197, 477
XGL_3D_CTX_SURF_FRONT_ILLUMINATION, 194, 410
XGL_3D_CTX_SURF_FRONT_LIGHT_COMPONENT, 410
XGL_3D_CTX_SURF_FRONT_LIGHT_COMPONENT, 194
XGL_3D_CTX_SURF_FRONT_SPECULAR, 194, 410
XGL_3D_CTX_SURF_FRONT_SPECULAR_COLOR, 194, 410
XGL_3D_CTX_SURF_FRONT_SPECULAR_POWER, 194, 410
XGL_3D_CTX_SURF_FRONT_TMAP, 467
XGL_3D_CTX_SURF_FRONT_TMAP_NUM, 468
XGL_3D_CTX_SURF_FRONT_TMAP_SWITCHES, 468
XGL_3D_CTX_SURF_FRONT_TRANSP, 195
XGL_3D_CTX_SURF_GEOM_NORMAL, 194, 410
XGL_3D_CTX_SURF_NORMAL_FLIP, 194, 411
XGL_3D_CTX_SURF_SILHOUETTE_EDGE_FLAG, 195, 259
XGL_3D_CTX_SURF_TMAP_PERSP_CORRECTION, 468
XGL_3D_CTX_SURF_TRANSP_BLEND_EQ, 195
XGL_3D_CTX_SURF_TRANSP_METHOD, 195
XGL_3D_CTX_VIEW_CLIP_PLUS_W_ONLY, 201
XGL_3D_CTX_Z_BUFFER_COMP_METHOD, 188
XGL_3D_CTX_Z_BUFFER_WRITE_MASK, 188
xgl_annotation_text, 173, 185, 318
XGL_BUFFER_SEL_ACCUM, 225
XGL_BUFFER_SEL_DISPLAY, 224
XGL_BUFFER_SEL_NONE, 224
XGL_CACHE_IS_EMPTY, 441
XGL_CACHE_USE_APPL_GEOM, 441

XGL_CGM_ABSTRACT, 70
XGL_CGM_BINARY, 69
XGL_CGM_CHARACTER, 69
XGL_CGM_CLEAR_TEXT, 69
XGL_CGM_DESCRIPTION, 69
XGL_CGM_ENCODING, 69
XGL_CGM_METRIC, 70
XGL_CGM_PICTURE_DESCRIPTION, 70
XGL_CGM_SCALE_FACTOR, 70
XGL_CGM_SCALE_MODE, 70
XGL_CGM_TYPE, 69
XGL_CGM_VDC_EXTENT, 71
xgl_close, 44
XGL_CMAP_COLOR_CUBE_SIZE, 131
XGL_CMAP_COLOR_MAPPER, 130
XGL_CMAP_COLOR_TABLE, 119, 500
XGL_CMAP_COLOR_TABLE_SIZE, 119
XGL_CMAP_DITHER_MASK, 132
XGL_CMAP_DITHER_MASK_N, 132
XGL_CMAP_INVERSE_COLOR_MAPPER, 130
XGL_CMAP_NAME, 132
XGL_CMAP_RAMP_LIST, 125
XGL_CMAP_RAMP_NUM, 125
XGL_COLOR_INDEX, 59
XGL_COLOR_RGB, 59
xgl_context_accumulate, 223
xgl_context_check_bbox, 182
xgl_context_clear_accumulation, 223
xgl_context_copy_buffer, 163, 213, 341
xgl_context_flush, 162
xgl_context_get_pixel, 163, 213
xgl_context_new_frame, 162
xgl_context_pop, 163, 164
xgl_context_post, 40, 73, 162
xgl_context_push, 163, 164
xgl_context_set_multi_pixel, 163, 213
xgl_context_set_pixel, 163, 213, 215
xgl_context_set_pixel_row, 163, 213
xgl_context_update_model_trans, 163
XGL_CTX_ARC_FILL_STYLE, 193
XGL_CTX_ATEXT_ALIGN_HORIZ, 199
XGL_CTX_ATEXT_ALIGN_VERT, 199
XGL_CTX_ATEXT_CHAR_HEIGHT, 199
XGL_CTX_ATEXT_CHAR_SLANT_ANGLE, 199
XGL_CTX_ATEXT_CHAR_UP_VECTOR, 199
XGL_CTX_ATEXT_PATH, 199
XGL_CTX_ATEXT_STYLE, 199
XGL_CTX_BACKGROUND_COLOR, 186
XGL_CTX_CLIP_PLANES, 200, 299
XGL_CTX_CURVE_APPROX_VAL, 235
XGL_CTX_DC_VIEWPORT, 200, 299
XGL_CTX_DEFERRAL_MODE, 40, 73, 164
XGL_CTX_DEVICE, 165
XGL_CTX_EDGE_AA_BLEND_EQ, 192
XGL_CTX_EDGE_AA_FILTER_SHAPE, 192
XGL_CTX_EDGE_AA_FILTER_WIDTH, 192
XGL_CTX_EDGE_ALT_COLOR, 192
XGL_CTX_EDGE_COLOR, 192
XGL_CTX_EDGE_JOIN, 192
XGL_CTX_EDGE_MITER_LIMIT, 192
XGL_CTX_EDGE_PATTERN, 192
XGL_CTX_EDGE_STYLE, 192
XGL_CTX_EDGE_WIDTH_SCALE_FACTOR, 192
XGL_CTX_FRONT_COLOR, 193, 410
XGL_CTX_GEOM_DATA_IS_VOLATILE, 166
XGL_CTX_GLOBAL_MODEL_TRANS, 200, 298
XGL_CTX_LINE_AA_BLEND_EQ, 189
XGL_CTX_LINE_AA_FILTER_SHAPE, 189

XGL_CTX_LINE_AA_FILTER_
WIDTH, 189
XGL_CTX_LINE_ALT_COLOR, 189, 362
XGL_CTX_LINE_CAP, 189
XGL_CTX_LINE_COLOR, 189, 362
XGL_CTX_LINE_COLOR_
SELECTOR, 189, 362
XGL_CTX_LINE_JOIN, 189
XGL_CTX_LINE_MITER_LIMIT, 189
XGL_CTX_LINE_PATTERN, 189
XGL_CTX_LINE_STYLE, 189
XGL_CTX_LINE_WIDTH_SCALE_
FACTOR, 190
XGL_CTX_LOCAL_MODEL_
TRANS, 200, 298
XGL_CTX_MARKER, 190, 376
XGL_CTX_MARKER_AA_BLEND_
EQ, 190
XGL_CTX_MARKER_AA_FILTER_
SHAPE, 190
XGL_CTX_MARKER_AA_FILTER_
WIDTH, 190
XGL_CTX_MARKER_COLOR, 190, 376
XGL_CTX_MARKER_COLOR_
SELECTOR, 190
XGL_CTX_MARKER_SCALE_
FACTOR, 190, 376
XGL_CTX_MAX_TESSELATION, 196
XGL_CTX_MAX_TESSELLATION, 191,
256
XGL_CTX_MC_TO_DC_TRANS, 200, 302
XGL_CTX_MIN_TESSELATION, 196
XGL_CTX_MIN_TESSELLATION, 191,
256
XGL_CTX_MODEL_TRANS, 200, 298
XGL_CTX_MODEL_TRANS_STACK_
SIZE, 164
XGL_CTX_NEW_FRAME_ACTION, 186
XGL_CTX_NEW_FRAME_PAINT_
TYPE, 99
XGL_CTX_NURBS_CURVE_
APPROX, 191, 235
XGL_CTX_NURBS_CURVE_APPROX_
VAL, 191
XGL_CTX_NURBS_SURF_
APPROX, 196, 254
XGL_CTX_NURBS_SURF_APPROX_
VAL, 196
XGL_CTX_NURBS_SURF_ISO_CURVE_
PLACEMENT, 196
XGL_CTX_NURBS_SURF_ISO_CURVE_
U_NUM, 196
XGL_CTX_NURBS_SURF_ISO_CURVE_
V_NUM, 196
XGL_CTX_NURBS_SURF_PARAM_
STYLE, 196, 257
XGL_CTX_PAINT_TYPE, 98
XGL_CTX_PLANE_MASK, 186
XGL_CTX_RASTER_FILL_STYLE, 186,
341
XGL_CTX_RASTER_FPAT, 186
XGL_CTX_RASTER_FPAT_
POSITION, 186
XGL_CTX_RASTER_SOURCE_
BUFFER, 163, 503
XGL_CTX_RASTER_STIPPLE_
COLOR, 186
XGL_CTX_RENDER_BUFFER, 163, 165,
187, 503
XGL_CTX_RENDERING_ORDER, 165
XGL_CTX_ROP, 186
XGL_CTX_SFONT_n, 198, 316
XGL_CTX_STEXT_AA_BLEND_EQ, 198
XGL_CTX_STEXT_AA_FILTER_
SHAPE, 198
XGL_CTX_STEXT_AA_FILTER_
WIDTH, 198
XGL_CTX_STEXT_ALIGN_HORIZ, 198
XGL_CTX_STEXT_ALIGN_VERT, 198
XGL_CTX_STEXT_CHAR_
ENCODING, 198, 331
XGL_CTX_STEXT_CHAR_EXPANSION_
FACTOR, 198
XGL_CTX_STEXT_CHAR_HEIGHT, 198

XGL_CTX_STEXT_CHAR_SLANT_ANGLE, 198
XGL_CTX_STEXT_CHAR_SPACING, 198
XGL_CTX_STEXT_CHAR_UP_VECTOR, 198
XGL_CTX_STEXT_COLOR, 198
XGL_CTX_STEXT_PATH, 198
XGL_CTX_STEXT_PRECISION, 198
XGL_CTX_SURF_AA_BLEND_EQ, 193
XGL_CTX_SURF_AA_FILTER_SHAPE, 193
XGL_CTX_SURF_AA_FILTER_WIDTH, 193
XGL_CTX_SURF_EDGE_FLAG, 192
XGL_CTX_SURF_FRONT_COLOR_SELECTOR, 193, 410
XGL_CTX_SURF_FRONT_FILL_STYLE, 193
XGL_CTX_SURF_FRONT_FPAT, 193
XGL_CTX_SURF_FRONT_FPAT_POSITION, 193
XGL_CTX_SURF_INTERIOR_RULE, 194
XGL_CTX_THRESHOLD, 186
XGL_CTX_VDC_MAP, 200, 299
XGL_CTX_VDC_ORIENTATION, 298
XGL_CTX_VDC_WINDOW, 200, 299
XGL_CTX_VIEW_CLIP_BOUNDS, 200, 298
XGL_CTX_VIEW_TRANS, 200, 298
XGL_DEFER_ASAP, 165
XGL_DEFER_ASTI, 165
XGL_DEV_COLOR_MAP, 58, 71
XGL_DEV_COLOR_TYPE, 58
XGL_DEV_CONTEXTS, 60
XGL_DEV_CONTEXTS_NUM, 59
XGL_DEV_MAXIMUM_COORDINATES, 60
XGL_DEV_REAL_COLOR_TYPE, 58
XGL_DMAP_TEXTURE_ORIENTATION_MATRIX, 477
XGL_DMAP_TEXTURE_PARAM_TYPE, 478
XGL_DMAP_TEXTURE_U_INDEX, 478, 479
XGL_DMAP_TEXTURE_V_INDEX, 478, 479
XGL_GCACHE_BYPASS_MODEL_CLIP, 440
XGL_GCACHE_DISPLAY_PRIM_TYPE, 440
XGL_GCACHE_DO_POLYGON_DECOMP, 440
XGL_GCACHE_FACET_LIST_LIST, 441
xgl_gcache_multi_elliptical_arc, 437
xgl_gcache_multi_simple_polygon, 437
xgl_gcache_multimarker, 437
xgl_gcache_multipolyline, 438
xgl_gcache_nurbs_curve, 438
XGL_GCACHE_NURBS_CURVE_MODE, 441
XGL_GCACHE_NURBS_SURF_MODE, 442
xgl_gcache_nurbs_surface, 438
XGL_GCACHE_ORIG_PRIM_TYPE, 440
xgl_gcache_polygon, 439
XGL_GCACHE_POLYGON_TYPE, 440
XGL_GCACHE_PT_LIST_LIST, 440
XGL_GCACHE_SHOW_DECOMP_EDGES, 440
xgl_gcache_stroke_text, 439
xgl_gcache_triangle_list, 439
xgl_gcache_triangle_strip, 439
XGL_GEOM_STATUS_FACING_FRONT, 182
xgl_image, 214, 341
xgl_inquire, 86
XGL_LIGHT_ATTENUATION_n, 409
XGL_LIGHT_COLOR, 409
xgl_light_copy, 409
XGL_LIGHT_DIRECTION, 409
XGL_LIGHT_POSITION, 409

XGL_LIGHT_SPOT_ANGLE, 409
XGL_LIGHT_SPOT_EXPONENT, 409
XGL_LIGHT_TYPE, 409
XGL_LPAT_DATA, 359
XGL_LPAT_DATA_SIZE, 359
XGL_LPAT_OFFSET, 360
XGL_MEM_RAS_IMAGE_BUFFER_ADDR, 64, 65
XGL_MEM_RAS_Z_BUFFER_ADDR, 65
xgl_mipmap_texture_build, 455
XGL_MIPMAP_TEXTURE_DEPTH, 458
XGL_MIPMAP_TEXTURE_HEIGHT, 458
XGL_MIPMAP_TEXTURE_LEVELS, 455
XGL_MIPMAP_TEXTURE_MEM_RAS_LIST, 457
XGL_MIPMAP_TEXTURE_WIDTH, 458
xgl_multi_elliptical_arc, 173, 184
xgl_multi_simple_polygon, 174
xgl_multiarc, 173, 184
XGL_MULTIARC_AF3D, 184
xgl_multicircle, 173, 184
XGL_MULTICIRCLE_AF3D, 184
XGL_MULTIELLIARC_AF3D, 184
xgl_multimarker, 174, 376
xgl_multipolyline, 174
XGL_MULTIRECT_AF3D, 184
xgl_multirectangle, 174, 184
xgl_nurbs_curve, 174, 233
xgl_nurbs_surface, 175, 249
xgl_object_create, 50, 268
xgl_object_destroy, 51
xgl_object_get, 51
xgl_object_set, 52
xgl_open, 43
XGL_PCACHE_CONTEXT, 432
xgl_pcache_display, 434
xgl_polygon, 175
xgl_quadrilateral_mesh, 175
XGL_RAS_ACCUM_BUFFER_DEPTH, 225
XGL_RAS_ACCUM_OP_DEST, 223
XGL_RAS_DEPTH, 60, 65
XGL_RAS_FILL_STENCIL, 341
XGL_RAS_HEIGHT, 60
XGL_RAS_RECT_LIST, 61
XGL_RAS_RECT_NUM, 61
XGL_RAS_SOURCE_BUFFER, 61
XGL_RAS_WIDTH, 60
XGL_RENDER_COMP_DIFFUSE_COLOR, 462
XGL_RENDER_COMP_FINAL_COLOR, 462
XGL_RENDER_COMP_REFLECTED_COLOR, 462
XGL_SFONT_COMMENT, 320
XGL_SFONT_DEFAULT_CHARACTER, 320
xgl_stroke_text, 175, 317
xgl_stroke_text_extent, 176, 320, 329
XGL_SYS_ST_ERROR_DETECTION, 44
XGL_SYS_ST_ERROR_NOTIFICATION_FUNCTION, 44, 45
XGL_SYS_ST_FONT_DIRECTORY, 314
XGL_SYS_ST_SFONT_DIRECTORY, 45
XGL_SYS_ST_VERSION, 45
XGL_TMAP_COORD_SOURCE, 469
XGL_TMAP_DOMAIN, 473
XGL_TMAP_PARAM_INFO, 470
XGL_TMAP_PARAM_TYPE, 470
XGL_TMAP_T0_INDEX, 470
XGL_TMAP_T1_INDEX, 470, 497
xgl_transform_copy, 273
xgl_transform_identity, 273
xgl_transform_invert, 274
xgl_transform_multiply, 274
xgl_transform_point, 275
xgl_transform_point_list, 275
xgl_transform_read, 271
xgl_transform_rotate, 270
xgl_transform_scale, 270
xgl_transform_translate, 271

xgl_transform_transpose, 275
xgl_transform_write, 273
xgl_transform_write_specific, 271
xgl_triangle_list, 176
xgl_triangle_strip, 176
XGL_WIN_RAS_BACKING_STORE, 40, 63
XGL_WIN_RAS_BUF_DISPLAY, 62
XGL_WIN_RAS_BUF_DRAW, 62
XGL_WIN_RAS_BUF_MIN_DELAY, 62
XGL_WIN_RAS_BUFFERS_ALLOCATED, 63
XGL_WIN_RAS_BUFFERS_REQUESTED, 62
XGL_WIN_RAS_DESCRIPTOR, 61
XGL_WIN_RAS_PIXEL_MAPPING, 61, 133
XGL_WIN_RAS_POSITION, 61
XGL_WIN_RAS_STEREO_MODE, 62
XGL_WIN_RAS_TYPE, 61
xgl_window_raster_resize, 74
XGLHOME, 45
XGLHOME environment variable, 45
Xlib
    remote rendering, 22
    XChangeWindowAttributes, 63
    XCreateColormap, 132
    XCreateWindow, 73
    XDrawString, 340
    XGL and Xlib, 22
    XSync, 40, 73
XSync, 40, 73
XView attributes
    CANVAS_AUTO_CLEAR, 38
    CANVAS_FIXED_IMAGE, 38
    CANVAS_RESIZE_PROC, 38

Z

Z-buffer, 65, 386
    write mask, 188, 506