

# XGL Device Pipeline Porting Guide

*Loadable Interfaces Version 4.0*

2550 Garcia Avenue  
Mountain View, CA 94043  
U.S.A.



© 1994 Sun Microsystems, Inc.  
2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

All rights reserved. This product and related documentation are protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or related documentation may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Portions of this product may be derived from the UNIX® and Berkeley 4.3 BSD systems, licensed from UNIX System Laboratories, Inc., a wholly owned subsidiary of Novell, Inc., and the University of California, respectively. Third-party font software in this product is protected by copyright and licensed from Sun's font suppliers.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the United States Government is subject to the restrictions set forth in DFARS 252.227-7013 (c)(1)(ii) and FAR 52.227-19.

The product described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

#### TRADEMARKS

Sun, the Sun logo, Sun Microsystems, Sun Microsystems Computer Corporation, SunSoft, the SunSoft logo, Solaris, SunOS, OpenWindows, DeskSet, ONC, ONC+, NFS, and XGL are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and certain other countries. UNIX is a registered trademark of Novell, Inc., in the United States and other countries; X/Open Company, Ltd., is the exclusive licensor of such trademark. OPEN LOOK® is a registered trademark of Novell, Inc. PostScript and Display PostScript are trademarks of Adobe Systems, Inc. All other product names mentioned herein are the trademarks of their respective owners.

All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. SPARCstation, SPARCserver, SPARCengine, SPARCstorage, SPARCware, SPARCcenter, SPARCclassic, SPARCcluster, SPARCdesign, SPARC811, SPARCprinter, UltraSPARC, microSPARC, SPARCworks, and SPARCcompiler are licensed exclusively to Sun Microsystems, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK and Sun™ Graphical User Interfaces were 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.

X Window System is a product of the Massachusetts Institute of Technology.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.



# *Contents*

---

Preface.....	xxi
New Features.....	xxv
<b>1. Introduction to XGL Loadable Interfaces.....</b>	<b>1</b>
Introduction to the XGL Product.....	1
Solaris Dynamic Linking .....	2
XGL Loadable Interfaces.....	2
Loadable Interface 1 (LI-1) .....	3
Loadable Interface 2 (LI-2) .....	3
Loadable Interface 3 (LI-3) .....	3
<b>2. Getting Started .....</b>	<b>7</b>
Before You Begin .....	8
Requirements .....	8
OpenWindows and XGL.....	9
Device Support for Multiple XGL Contexts.....	9
Device Support for Backing Store .....	10

---

XGL Architecture from the Pipeline Point of View . . . . .	11
More on Device Pipelines . . . . .	11
Services the XGL Core Provides the Device Pipeline . . . . .	12
Porting Task . . . . .	14
Choosing a Loadable Interface Level . . . . .	14
A Quick Look at Implementing an LI-1 Primitive . . . . .	16
Testing Your Implementation . . . . .	20
Calling the Software Pipeline . . . . .	20
Device Pipeline Options to Rendering Calls . . . . .	20
What Else You Should Know . . . . .	21
Accessing External Files at Runtime . . . . .	21
Directory Structure for the XGL DDK . . . . .	22
Error Reporting for XGL Device Pipelines . . . . .	23
<b>3. Pipeline Framework . . . . .</b>	<b>29</b>
Overview of the Pipeline Framework . . . . .	30
Setting Up the Pipeline Framework . . . . .	32
Defining xgl_create_PipeLib() . . . . .	32
Defining the Device Pipeline Library Class . . . . .	33
Defining the Device Pipeline Manager Class . . . . .	36
Defining the Device Pipeline Device Class . . . . .	38
Defining the Device Pipeline-Context Class . . . . .	43
Naming Your Device Pipeline . . . . .	51
Optional Functions in Device-Dependent Classes . . . . .	52
Overridable Functions in DpDev.h . . . . .	52

---

Overridable Functions in DpDevRaster.h . . . . .	53
Overridable Functions in DpDevWinRas.h . . . . .	54
Overridable Functions in DpDevMemRas.h . . . . .	56
What Else You Should Know . . . . .	58
How a Device Pipeline Is Loaded . . . . .	58
Device Pipeline Objects for Multiple Processes . . . . .	59
Adding Member Data to a Pipeline Class . . . . .	61
Versioning . . . . .	63
Backing Store Support in the Pipeline Classes . . . . .	65
Quick Reference Chart of Overridable Functions . . . . .	68
<b>4. Internal Data Storage . . . . .</b>	<b>73</b>
Internal Data Types . . . . .	74
Accessing Data at the LI-1 Layer . . . . .	75
Accessing Application Data . . . . .	75
Accessing Facet Data . . . . .	76
Point Lists with Data Mapping Values . . . . .	77
Data Access for DMA Devices . . . . .	78
How Data Is Stored by the Software Pipeline . . . . .	79
Data Storage in the XglLevel Object . . . . .	80
LI-2 Point Data . . . . .	82
Accessing Data at the LI-2 Layer . . . . .	82
Pipeline Interfaces to XglPrimData and XglLevel Data . . .	84
Conic and Rectangle Data . . . . .	85
Accessing Rectangle Data from XglRectData . . . . .	85

---

Accessing Conic Data from XglConicData . . . . .	86
Pipeline Interfaces to XglConicData and XglRectData . . . .	88
Pixel Data . . . . .	89
Using PixRects . . . . .	90
PixRect Interfaces. . . . .	91
<b>5. Handling Changes to Object State. . . . .</b>	<b>95</b>
State Changes and the Device Pipeline. . . . .	96
Getting Attribute Values from the Context Object. . . . .	96
When the Device Associated with a Context Is Changed .	98
Getting Attribute Values from Objects Other Than the Context	99
Handling Derived Data Changes. . . . .	105
Getting Stroke Attribute Values from the Stroke Group Object	106
Example of Device Pipeline Use of Stroke Groups . . . . .	107
Rendering Multipolylines. . . . .	109
Flag Mask and Expected Flag Value . . . . .	111
DC Offset . . . . .	112
Design Issues . . . . .	114
Deciding to Reject a Primitive . . . . .	114
Handling Context Switches . . . . .	115
Handling Changes in LI Levels . . . . .	116
Partial Rendering of a Primitive . . . . .	117
<b>6. Getting Information from XGL Objects . . . . .</b>	<b>119</b>
What You Should Know About XGL Attribute Values . . . . .	120
Naming Conventions for Internal Attributes. . . . .	121

---

Context Attributes and LI Layers . . . . .	122
Getting Attribute Values from the Context. . . . .	123
Getting Attribute Values from Other Objects. . . . .	123
Getting Information from a Transform Object . . . . .	125
Getting Attribute Values From the Stroke Group Object . . . . .	125
Non-API Interfaces Provided in API Objects. . . . .	127
Context Interfaces . . . . .	127
Context 2D Interfaces . . . . .	128
Context 3D Interfaces . . . . .	129
Data Map Texture Interfaces . . . . .	130
Device Interfaces . . . . .	131
Light Interfaces . . . . .	132
Line Pattern Interfaces . . . . .	132
Marker Interfaces. . . . .	133
MipMap Texture Interfaces. . . . .	133
Raster Interfaces. . . . .	134
Texture Map Interfaces . . . . .	134
Window Raster Interfaces. . . . .	135
Memory Raster Interfaces. . . . .	135
Stroke Font Interfaces . . . . .	136
Transform Interfaces and Flags . . . . .	137
Getting Information From the Device Object. . . . .	145
Color Map Interfaces. . . . .	145
<b>7. View Model Derived Data. . . . .</b>	<b>149</b>

---

Overview of View Model Derived Data . . . . .	150
Design Goals of Derived Data . . . . .	151
Derived Data Items . . . . .	154
Coordinate Systems and Transforms . . . . .	154
Other Derived Items. . . . .	156
Overview of Derived Data's Implementation . . . . .	157
Accessing Derived Data . . . . .	158
Registration of Concerns. . . . .	159
Bit Definitions for the View Flag . . . . .	161
Determining Whether Derived Items Have Changed. . . . .	163
Messages. . . . .	163
The Composite . . . . .	164
Detecting Changes With the Composite . . . . .	164
Setting the Composite . . . . .	165
Clearing the Composite . . . . .	165
Detecting Changes to Individual Derived Items . . . . .	166
Getting Derived Items. . . . .	168
Getting Derived Transforms . . . . .	169
Getting Boundaries. . . . .	170
Getting 3D Viewing Flags . . . . .	171
Getting Lights . . . . .	172
Getting Eye Positions or Vectors. . . . .	173
Getting Model Clip Planes. . . . .	174
Getting Depth Cue Reference Planes . . . . .	175



---

Example of Detecting Changes and Getting Derived Items. . .	175
Current Coordinate System . . . . .	180
<b>8. Window System Interactions . . . . .</b>	<b>183</b>
Overview of the XglDrawable . . . . .	184
Services Provided by XglDrawable Class. . . . .	185
A Typical Scenario of Drawable Creation and Use . . . . .	186
What You Should Know About Locking the Window . . . . .	187
Drawable Interfaces for the Pipeline. . . . .	189
Obtaining Information During Pipeline Initialization . . . .	190
Accessing Dynamic Information Through the Drawable. .	191
Managing Window System Resources . . . . .	195
Managing Software Cursors. . . . .	196
Description of Drawable Interfaces. . . . .	197
XglDrawable Functions for the Device Pipeline . . . . .	197
XglDrawable Functions Used by the XGL Core Only. . . . .	203
Window System Dependencies . . . . .	205
<b>9. Writing Loadable Interfaces . . . . .</b>	<b>207</b>
What You Need to Know about the Loadable Interfaces . . . . .	208
Overview List of Loadable Pipeline Interfaces . . . . .	209
Deciding Which Interfaces to Implement. . . . .	211
Input Data for LI-2 and LI-3. . . . .	214
Picking. . . . .	214
Hints for Rendering Transparent 3D Surfaces . . . . .	215
Calling the Software Pipeline for Texture Mapping . . . . .	216

---

Antialiasing and Dithering . . . . .	218
Mapping of API Primitive Calls to LI-1 Functions . . . . .	218
What You Should Know about the Software Pipeline . . . . .	220
Software Pipeline Multiplexing . . . . .	220
Software Pipeline Backing Store . . . . .	221
Surface Color in the Software Pipeline . . . . .	221
Texture Mapping in the Software Pipeline . . . . .	222
LI-1 Functions . . . . .	223
About the LI-1 Layer . . . . .	223
LI-1 Operations in the Software Pipeline . . . . .	223
Mapping of LI-1 to LI-2 Functions in the Software Pipeline . . . . .	224
LI-1 Attributes . . . . .	226
li1AnnotationText() - 2D/3D . . . . .	227
li1DisplayGcache() - 2D/3D . . . . .	229
li1MultiArc() - 2D . . . . .	250
li1MultiArc() - 3D . . . . .	252
li1MultiCircle() - 2D . . . . .	254
li1MultiCircle() - 3D . . . . .	256
li1MultiEllipticalArc() - 3D . . . . .	258
li1MultiMarker() - 2D . . . . .	260
li1MultiMarker() - 3D . . . . .	261
li1MultiPolyline() - 2D . . . . .	263
li1MultiPolyline() - 3D . . . . .	264
li1MultiRectangle() - 2D . . . . .	266

---

li1MultiRectangle() - 3D .....	267
li1MultiSimplePolygon() - 2D .....	269
li1MultiSimplePolygon() - 3D .....	270
li1NurbsCurve() - 2D.....	272
li1NurbsCurve() - 3D.....	274
li1NurbsSurf() - 3D .....	276
li1Polygon() - 2D .....	278
li1Polygon() - 3D .....	280
li1QuadrilateralMesh() - 3D.....	282
li1StrokeText() - 2D/3D.....	283
li1TriangleList() - 3D .....	285
li1TriangleStrip() - 3D .....	288
li1Accumulate() - 3D .....	290
li1ClearAccumulation() - 3D .....	292
li1CopyBuffer() - 2D/3D.....	293
li1Flush() - 2D/3D.....	297
li1GetPixel() - 2D/3D .....	298
li1Image() - 2D/3D .....	299
li1NewFrame() - 2D/3D .....	301
li1PickBufferFlush() - 2D/3D.....	303
li1SetMultiPixel() .....	305
li1SetPixel() - 2D/3D.....	306
li1SetPixelRow() - 2D/3D.....	307
LI2 Functions .....	308

---

About the LI-2 Layer . . . . .	308
LI-2 Surface Attributes . . . . .	308
Mapping of LI-2 Functions to LI-3 Functions in the Software Pipeline . . . . .	310
li2GeneralPolygon() - 2D/3D . . . . .	312
li2MultiDot() - 2D/3D . . . . .	313
li2MultiEllipse() - 2D . . . . .	314
li2MultiEllipticalArc() - 2D . . . . .	315
li2MultiPolyline() - 2D . . . . .	316
li2MultiPolyline() - 3D . . . . .	318
li2MultiRect() - 2D . . . . .	320
li2MultiSimplePolygon() - 2D . . . . .	321
li2MultiSimplePolygon() - 3D . . . . .	322
li2TriangleList() - 3D . . . . .	323
li2TriangleStrip() - 3D . . . . .	324
LI-3 Functions. . . . .	325
About the LI-3 Layer . . . . .	325
Notes on Implementing LI-3 Functions . . . . .	326
li3Begin() and li3End() - 2D/3D . . . . .	328
li3CopyFromDpBuffer() - 2D/3D . . . . .	329
li3CopyToDpBuffer() - 2D . . . . .	330
li3CopyToDpBuffer() - 3D . . . . .	331
li3MultiDot() - 2D . . . . .	333
li3MultiDot() - 3D . . . . .	334

---

li3Vector() - 2D .....	336
li3Vector() - 3D .....	338
li3MultiSpan() - 2D .....	341
li3MultiSpan() - 3D .....	343
<b>10. Utilities .....</b>	<b>347</b>
RefDpCtx .....	348
Using RefDpCtx .....	349
RefDpCtx Interfaces .....	351
3D Utilities .....	352
Bounding Box Utilities .....	397
Copy Buffer Utilities .....	399
Polygon Classification Utilities .....	403
Polygon Decomposition Utilities .....	405
<b>A. Performance Tuning .....</b>	<b>409</b>
Finding the Performance Critical Paths .....	410
At-a-Glance Comparison of Performance Tools .....	412
Recommendations for Performance Tools .....	413
Selecting Good Benchmarks .....	413
Tuning Performance Critical Paths .....	415
Locating the Central Body of Code .....	415
Changing the Underlying Algorithm .....	415
Tuning at the Assembly Language Level .....	416
Tips and Techniques for Faster Code .....	416
<b>B. Changes to the XGL Graphics Porting Interface .....</b>	<b>439</b>

---

Changes in Rendering Architecture . . . . .	439
Changes in State Handling . . . . .	442
Application Data Passed Directly to Pipelines . . . . .	443
<b>C. Accelerating NURBS Primitives . . . . .</b>	<b>445</b>
Index . . . . .	447

## *Figures*

---

Figure 1-1	XGL Loadable Interface Layers . . . . .	4
Figure 2-1	Basic View of XGL Architecture . . . . .	11
Figure 2-2	High-Level View of the XGL Primitive Call Processing . . . . .	13
Figure 2-3	XGL DDK Directory Structure. . . . .	22
Figure 3-1	Device Pipeline Framework Classes. . . . .	31
Figure 3-2	Pipeline Objects for a Single Application. . . . .	59
Figure 3-3	Pipeline Objects for an Application on Multiple Frame Buffers	60
Figure 3-4	Pipeline Objects for Two Applications. . . . .	60
Figure 3-5	Pipeline Objects for Applications on Multiple Frame Buffers	61
Figure 4-1	Flow of Application Data Through the LI-1 Primitive. . . . .	75
Figure 4-2	Level Objects Created by Software Pipeline Processing . . . . .	79
Figure 4-3	Flow of Point Data Through XglPrimData and XglLevel . . . . .	80
Figure 4-4	Base/Offset Data Storage in XglLevel . . . . .	80
Figure 4-5	Base/Offset Data When the Point Data Has Changed . . . . .	81
Figure 4-6	XglPixRect Class Hierarchy . . . . .	90
Figure 5-1	Attribute Processing Using the Stroke Group. . . . .	108

---

Figure 6-1	DI and Dp Object Relationships .....	120
Figure 6-2	Device Pipeline and Layered Attributes .....	122



## *Tables*

---

Table 3-1	Device-Dependent Overridable Functions . . . . .	41
Table 3-2	Summary of Optional and Required Pipeline Functions . . . .	68
Table 4-1	XglPrimData Interfaces. . . . .	84
Table 4-2	XglLevel . . . . .	84
Table 4-3	XglConicData Interfaces . . . . .	88
Table 4-4	XglConicList2d Interfaces . . . . .	88
Table 4-5	XglRectList2d and XglRectList3d . . . . .	89
Table 4-6	XglPixRect Interfaces . . . . .	92
Table 4-7	XglPixRectMem Interfaces . . . . .	93
Table 4-8	XglPixRectMemAllocated Interfaces . . . . .	93
Table 4-9	XglPixRectMemAssigned Interfaces. . . . .	94
Table 5-1	Object Messages. . . . .	100
Table 5-2	Stroke Table Flag Mask and Expected Flag Mask Values . . . .	111
Table 5-3	Stroke Group DC Offset Values . . . . .	113
Table 6-1	Getting Information from Xgl Objects . . . . .	124
Table 6-2	XGLI_TRANS_SINGULAR . . . . .	137

---

Table 7-1	Derived Data 2D Coordinate Systems . . . . .	155
Table 7-2	Derived Data 3D Coordinate Systems . . . . .	155
Table 7-3	Other Items in Derived Data . . . . .	156
Table 7-4	View Model Derived Data Classes . . . . .	157
Table 7-5	Bits for the View Flag . . . . .	163
Table 7-6	Functions to Return the Change Status of Derived Items . . . .	167
Table 7-7	Functions for Getting Derived Transforms . . . . .	170
Table 7-8	Functions for Getting Boundaries . . . . .	170
Table 8-1	Drawable Subclasses . . . . .	184
Table 8-2	Drawable Interfaces Used During Pipeline Initialization . . . .	190
Table 8-3	Drawable Interfaces Used During Rendering . . . . .	193
Table 8-4	Drawable Interfaces Used for Allocating Resources . . . . .	195
Table 9-1	List of Loadable Pipeline Interfaces . . . . .	209
Table 9-2	LI1 to LI2 Dependencies . . . . .	212
Table 9-3	LI-2 to LI-3 Dependencies . . . . .	213
Table 9-4	Mapping of 2D Primitives to 2D LI-1 Functions . . . . .	218
Table 9-5	Mapping of 3D API Primitives to 3D LI-1 Functions . . . . .	219
Table 9-6	Mapping of API Utility Functions to LI-1 Functions . . . . .	220
Table 9-7	Mapping of 2D LI-1 Functions to LI-1 and LI-2 Functions . . .	224
Table 9-8	Mapping of 3D LI-1 Functions to LI-1 or LI-2 Functions . . . .	225
Table 9-9	Gcache Interfaces . . . . .	244
Table 9-10	Surface Attributes at LI-2 . . . . .	309
Table 9-11	Mapping of 2D LI-2 Functions to LI-2 or LI-3 Functions . . . .	310
Table 9-12	Mapping of 3D LI-2 Functions to LI-2 and LI-3 Functions . . .	310
Table 9-13	LI-3 Primitive Functions . . . . .	325

---

Table A-1	Comparing Applications Used to Gather Profile Information	412
Table A-2	Compiler Options .....	437
Table B-1	Changed Utilities for XGL 3.1 .....	443



## *Preface*

---

The *XGL Device Pipeline Porting Guide* documents the interfaces and concepts required to write graphics device handlers (otherwise known as *loadable device pipelines*) for XGL™. These dynamically loadable modules enable applications running on XGL to exploit fully the capabilities of graphics accelerators present at runtime.

### *Who Should Use This Book*

This document is intended for implementors of XGL device pipelines. It is assumed that the reader is familiar with the C and C++ language and with the ideas of classes and class inheritance in C++.

### *How This Book Is Organized*

This manual is organized into 10 chapters and several appendixes.

**Chapter 1, “Introduction to XGL Loadable Interfaces”** presents an introduction to the XGL product and an overview of the three levels of the XGL graphics porting interface.

**Chapter 2, “Getting Started”** provides an overview of the porting process.

**Chapter 3, “Pipeline Framework”** presents information on the objects that connect XGL device-independent code with the device pipeline code.

---

**Chapter 4, “Internal Data Storage”** discusses the data structures used to represent internal data in XGL.

**Chapter 5, “Handling Changes to Object State”** describes how a device pipeline gets information about changes to XGL state.

**Chapter 6, “Getting Information from XGL Objects”** describes how a device pipeline gets information on XGL state.

**Chapter 7, “View Model Derived Data”** describes how a device pipeline gets information about changes to view model data.

**Chapter 8, “Window System Interactions”** provides information on the relationship between XGL, DGA, the window system, and the device pipelines, and discusses the mechanism by which XGL communicates with the window system.

**Chapter 9, “Writing Loadable Interfaces”** describes the the complete set of loadable interfaces.

**Chapter 10, “Utilities”** provides information on the XGL utilities.

**Appendix A, “Performance Tuning”** provides information on how to tune your code for optimum performance.

**Appendix B, “Changes to the XGL Graphics Porting Interface”** provides information on changes in the graphics porting interface between the previous release and the current release.

**Appendix C, “Accelerating NURBS Primitives”** provides references for XGL NURBS algorithms.

## *Related Books*

For information on the XGL architecture and the object-oriented design of the loadable pipelines, see the following document:

- *XGL Architecture Guide*, part number 801-6675-10.

For information on the XGL test suite, see:

- *XGL Test Suite User’s Guide* part number 801-6762-10.

---

For information on the XGL product, see the following documents:

- *XGL Programmer's Guide*, part number 801-6670-10.
- *XGL Reference Manual*, part number 801-6671-10.

## What Typographic Changes and Symbols Mean

Table P-1 describes the type changes and symbols 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 <code>.login</code> file. Use <code>ls -a</code> to list all files. <code>system%</code> You have mail.
<i>AaBbCc123</i>	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in <i>User's Guide</i> . These are called <i>class</i> options. You <i>must</i> be root to do this.

## XGL Sample Device Handler Usage Rights and Restrictions

The sample device handler code provided with the current XGL DDK package and the source code excerpts presented in this documentation are intended to help you create an XGL loadable pipeline for your product. You can copy, duplicate, or modify any section of the source code, and redistribute object code, as long as its usage is to create a loadable pipeline for XGL. This excludes authorization to redistribute source code created by using the source code information provided by SunSoft. Any other use is therefore prohibited and requires explicit agreements with SunSoft.





## *New Features*

---

This section lists new features in the XGL 3.1 version of the graphics porting interface.

### *Optimization of Device-Independent Operations*

The XGL 3.1 release of the graphics porting interface (GPI) includes a number of changes aimed at improving XGL's low batching factor performance. The two main goals for the optimization effort were to minimize the device-independent overhead for each graphics primitive call and to simplify the interface between the device pipelines and the device-independent code.

XGL's architecture was changed to implement the performance improvements. The new architecture differs from the previous architecture in two major ways:

- Device-independent code uses function pointers to call the device pipeline renderers directly from the API wrapper rather than through the interface manager. The functionality provided by the interface manager in the previous releases is implemented in other ways at this release, and the interface manager object is no longer present.
- Device-independent code notifies the device pipeline of Context attribute changes immediately (non-lazily) except for transform changes (which are still lazy evaluated). As a result, the update table objects are no longer present.

---

As a result of the changes to the XGL architecture, existing XGL device pipelines need to be modified. For detailed information on required updates to the pipelines, see Appendix B, “Changes to the XGL Graphics Porting Interface”.

# *Introduction to XGL Loadable Interfaces*

---

1 

## *Introduction to the XGL Product*

The XGL product is a foundation graphics library that provides geometry graphics support for Solaris®-based applications. The XGL library has two sets of interfaces: an application programming interface (API) and a graphics porting interface (GPI).

The XGL API provides application developers with immediate-mode rendering, a rich set of graphics primitives, view and modeling transforms, and separate, complete 2D and 3D rendering pipelines. Standard features include 2D and 3D primitive support; depth cueing, lighting, and shading; non-uniform B-spline curve and surface support; and direct and indirect color model support. Advanced features include transparency, antialiasing, texture mapping, stereo, and accumulation buffer for motion blur and other special effects. Application developers and developers of other graphics APIs can port their applications to XGL and take advantage of Solaris dynamic linking to provide portable shrink-wrapped applications that run on any graphics device supported in the Solaris environment. The XGL API is provided as part of the Solaris Developer's Kit; for more information on the XGL API, see the *XGL Reference Manual* and the *XGL Programmer's Guide*.

The XGL GPI is a device-level interface that defines the mapping of XGL device handlers to underlying hardware. Hardware vendors that write XGL device handlers can build graphics devices that support any binary XGL application. The XGL architecture provides open, well-defined interfaces that facilitate the task of implementing device handlers.

## *Solaris Dynamic Linking*

The Solaris 2.x operating system includes support for dynamic linking of shared libraries. A shared library is a library that can be dynamically linked during the running of the application. Under dynamic linking, the contents of the shared library are mapped into the application's virtual address space at runtime. References by the application to the functions in the shared library are resolved as the program executes.

The Solaris environment provides mechanisms to dynamically load both kernel device drivers and user process shared libraries. These facilities allow a hardware vendor to incorporate a new graphics accelerator into the Solaris environment by providing a dynamically loadable kernel device driver and an XGL device handler.

## *XGL Loadable Interfaces*

The XGL GPI consists of three layers of device pipeline interfaces. Each layer defines a set of rendering tasks that must be accomplished before proceeding to the next layer in the pipeline. More complex operations, such as transformations, lighting, and clipping, are performed in the uppermost layer; less complex operations, such as scan conversion, are performed in the lower layers. You can implement GPI functions at varying layers to tailor a port for your device.

The XGL GPI includes a complete software implementation of the top two layers of the pipeline for most primitives. The lowest layer, which is responsible for writing pixels to the device, is device dependent and has not been included in the software implementation.

You can choose a layer for your device handler based on the functionality of your device and let the XGL software implementation handle the rendering of functionality not accelerated by the device. The selection of the interface layer to port to can be made for each graphics primitive. Before each layer calls the layer below it, a device handler has the opportunity to either interpose its own code for a particular primitive or let the XGL-supplied software implementation perform the rendering tasks. Thus, for each primitive, a device handler can be called at the layer for which it is best adapted.

The functions comprising the software implementation and the device-dependent functions that replace them are grouped into separate dynamically loadable libraries. The set of device-dependent functions is called the device pipeline. The complete software implementation is called the software pipeline. At runtime, when an application program calls a primitive, the XGL device-independent code decides whether to render using the software pipeline or the device pipeline. This decision depends on the capabilities of the hardware and on the current XGL primitive and the current graphics state as defined by XGL's attributes.

### *Loadable Interface 1 (LI-1)*

The topmost layer is called Loadable Interface 1, or LI-1. This layer is directly below the XGL API. An LI-1 device handler is responsible for all aspects of drawing an XGL primitive, including transformation, clipping (view and model), lighting, and depth cueing. Devices that port to this layer for some or all of the XGL primitives are responsible for all operations required for rendering, including scan conversion and rendering of pixels. Although this is the most difficult layer to port to, a port to LI-1 enables full acceleration on a graphics device.

### *Loadable Interface 2 (LI-2)*

The second layer, LI-2, is responsible primarily for scan converting more complex primitives like polygons and polylines. Porting to this layer assumes some responsibility for rendering (especially if the hardware supports scan-conversion of primitives) but leaves the processing of the geometry (transformation, clipping, and so on) to the XGL software version of the layer above.

### *Loadable Interface 3 (LI-3)*

The lowest layer in the device pipeline, LI-3, is responsible for rendering pixels and vectors individually, or in spans. If you port to this layer, you need only implement vectors, span, and dot renderers. All other operations needed to process an API primitive and reduce it to this level are provided by XGL's default software implementation.

Because writing pixels to the frame buffer is device dependent, the software pipeline does not implement the LI-3 layer. Device handlers for new devices must implement LI-3 functions. To assist you with an LI-3 port, XGL provides utilities that perform pixel operations. You can call these utilities in place of writing a device-specific LI-3 layer.

Figure 1-1 illustrates the layers of the device pipeline and software pipeline as well as some of the components of the XGL device-independent code.

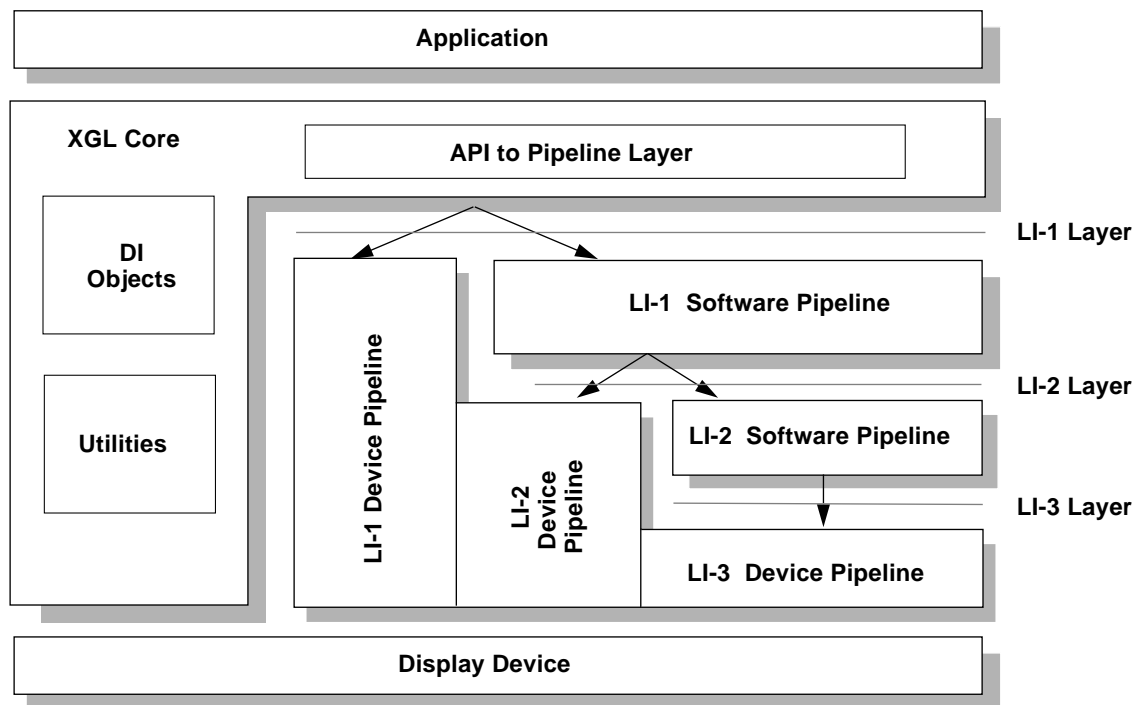


Figure 1-1 XGL Loadable Interface Layers

As mentioned above, the decision as to which layer to port to can be made on a per-primitive basis. For example, if a particular hardware device can render polylines but not polygons, a device handler for that device might implement the polyline primitive at LI-1 and let the XGL software pipeline render the polygons. At any time, a device handler can override the default software

interface provided by XGL. This choice is dynamic and is flexible enough to permit a variety of hardware devices to fully utilize their capabilities to draw XGL primitives.

---

**Note** – Currently, the XGL graphics porting interface is unstable. It is possible that this interface could change in the future in ways that could require changes in device pipelines.

---





## *Getting Started*

---



This chapter presents information that you will need as you write your device handler. The following topics are covered:

- Issues that you need to consider before beginning your port
- Brief description of the XGL architecture as it relates to the device handler
- Information on the porting task, including a summary of how to write an LI-1 device pipeline
- Information on calling the XGL software pipeline from a device handler
- Directions on how to add error processing to a device handler

## *Before You Begin*

Before beginning your device pipeline, check the requirements described below and plan how you will implement support for multiple XGL Context objects. You will also need to determine whether your device will support backing store.

### *Requirements*

When porting a device to XGL and the Sun platform, you must provide support for the operating system kernel and the window system in addition to providing the XGL loadable pipeline for the hardware. Thus, with your device pipeline, you must supply the following:

- A hardware interface from your device to the host platform using the system bus and a boot PROM to identify and initialize the device. For SPARCstation™ and SPARC-compatible systems, the hardware interface is the SBus. For information on writing an SBus interface, see *Writing FCode Programs* and the *OpenBoot Command Reference Manual*.
- A kernel device driver for the device. For information on developing a device driver, see *Writing Device Drivers*.
- A DDX handler for the X11 server for the graphics accelerator. Depending on your hardware, you may also need to port the device-dependent portions of Sun's Direct Graphics Access (DGA) mechanism to your hardware. For information, see the *OpenWindows Server Device Developer's Guide*.

This documentation and the XGL graphics porting interface is part of the Solaris Driver Developer's Kit (DDK). The Solaris DDK describes the interfaces between the Solaris environment and the hardware platform. The DDK includes information on the Solaris VISUAL environment, Solaris graphics and imaging foundation libraries, the Solaris X11 server, kernel device drivers for graphics and imaging devices, and the physical connections between graphics devices and Solaris platforms. The DDK also includes header files and sample code to help you develop a graphics accelerator and integrate it into the Solaris environment.

## *OpenWindows and XGL*

The OpenWindows™ environment includes Sun's Direct Graphics Access (DGA) technology, which arbitrates access to the display screen between XGL and the window system. DGA defines a protocol between the client application (XGL in this case) and the X11 window server that enables both the application and server to share the underlying graphics hardware. When the application is running on the same machine as the OpenWindows server and the hardware has DGA support, XGL uses DGA to synchronize on-screen drawing with the server. For local rendering, DGA allows XGL to send commands directly to the accelerator or frame buffer, substantially improving performance. When the XGL client program is running remotely, XGL uses Xlib or PEXlib to do all rendering.

As part of your port of the OpenWindows server to your device, you may need to port DGA as well. Your device-specific version of DGA enables XGL to render directly to your device.

## *Device Support for Multiple XGL Contexts*

A hardware device can be used by many different graphics rendering processes at once. At a minimum, the device will be used by the display server and one XGL client, and there may be other libraries or additional XGL clients using the device as well. Each task will maintain a current state or *context*, such as line color. Since the device is being shared by multiple users, the state must be current for each user before drawing can take place. Thus, your hardware resources must be able to support multiple contexts.

The term *context* refers to a set of state information that controls an executing entity. The use of this term can become confusing at times because it can refer to any one of the following:

Hardware context	State information that defines rendering characteristics on graphics accelerators, for example line color or raster operation register values.
Process context	State information that controls a UNIX process, such as the program counter, the signal mask, or file descriptors. This state also includes memory mapping information for devices.

XGL Context	State information that defines the rendering of XGL primitives, such as line color or transforms.
-------------	---

Because graphics hardware support for context switching is device dependent, state changes resulting from intraprocess switching of XGL Contexts must be managed within the device pipeline. Thus, early in the device pipeline design phase, you should consider how your device pipeline will support multiple XGL Contexts within a single process.

Also, multiple processes can access your hardware simultaneously. It is important to define how your device will allocate and share its resources among different processes and different windows within a process. Efficient sharing of hardware resources will enable your pipeline to make better use of the XGL architecture.

### *Device Support for Backing Store*

Backing store is a mechanism that saves the obscured portions of a window so that the window can be refreshed quickly when it becomes visible again. A *backing store* is off-screen memory that reflects the contents of the display buffer. This memory is used by the server to automatically restore previously obscured areas of the display during an expose event. Backing store can be handled by your graphics device or by XGL.

If you can use your graphics device to implement backing store, the device must be able to render into off-screen memory, and in your implementation of the OpenWindows server, you need to enable the backing store feature. A request for backing store support from the server will then allocate backing store memory from your hardware.

If your device does not support backing store, you can request that the server and XGL handle it instead. To use XGL for backing store support, you must implement a small set of device-dependent functions in the pipeline. If your device has a software Z-buffer or accumulation buffer, then the contents of these buffers must be shared with the backing store to keep the buffers and their backing store counterparts synchronized, since the server only repairs damage to the display buffer. See page 65 for more information on using XGL to support backing store, and see the *XGL Architecture Guide* for information on the architecture of backing store.

## *XGL Architecture from the Pipeline Point of View*

The XGL architecture defines two basic components: the device-independent *core* and the device-dependent loadable device pipelines. The XGL core functions as the interface between the application program and the device pipelines. The pipelines turn geometric primitives and their state attributes into pixel data that is displayed on a graphics hardware device or written into memory. Figure 2-1 illustrates these basic components:

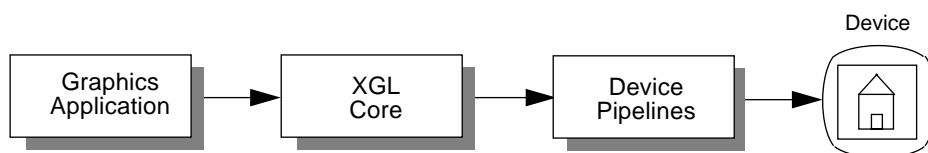


Figure 2-1 Basic View of XGL Architecture

A device pipeline can also be thought of as having two components: a set of objects that constitute the abstract XGL Device, and the loadable interfaces that send the application data to the hardware. The set of *device* objects link the loadable device pipeline with the XGL core and serve as a framework connecting the device-independent code with the device pipeline rendering code.

### *More on Device Pipelines*

Graphics applications perform operations in particular coordinate systems. These operations include clipping, lighting, projection, depth cueing, and mapping to a viewport. A device pipeline consists of a series of transformations from coordinate system to coordinate system.

The conceptual pipeline, which is independent of the implementation across graphics platforms, is the sequence of transformations and operations for a graphics primitive. The actual implementation within a device pipeline for a particular device may reorder the operations to enhance performance while preserving a representation of the final geometric description. However, the device may only be capable of increasing the performance under certain conditions. For other conditions, the device pipeline calls the XGL software pipeline, which can handle any valid combination of conditions.

A device pipeline can be written at three layers, LI-1, LI-2, and LI-3. Chapter 1, “Introduction to XGL Loadable Interfaces” briefly describes these layers. A graphics device with a high degree of functionality may choose to implement a complete primitive at the LI-1 level, in effect bypassing the lower levels. For example, a device may implement the triangle strip primitive at the LI-1 level by executing all of the operations of the rendering pipeline on the device. When a device is unable to handle some situations, for example, dithering with a color cube, it can fall back to the software pipeline for the function specific to the situation.

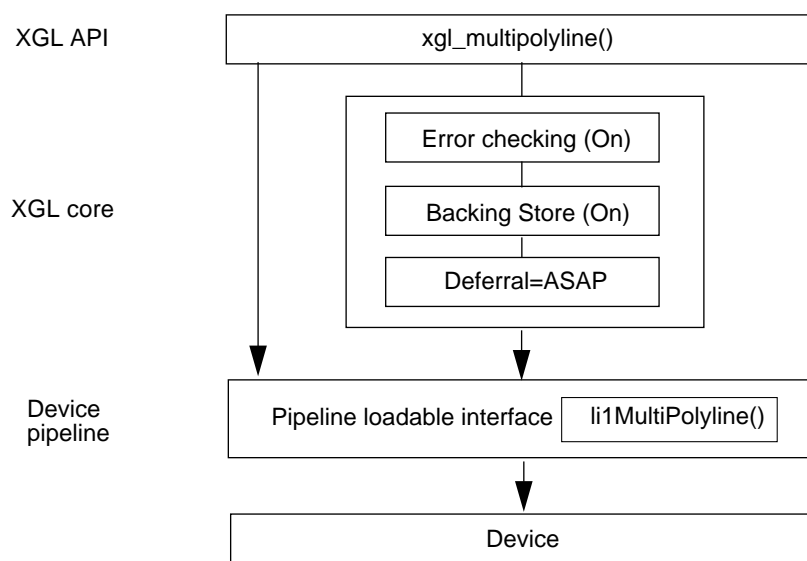
If you have a simple frame buffer and want to do a minimal amount of work to write a device handler for the device, you would choose to port to the LI-3 level. The pipeline for a simple frame buffer relies on the software pipeline geometric and rendering functions to feed the pixel-level interface at the LI-3 level of the device pipeline.

## *Services the XGL Core Provides the Device Pipeline*

The XGL device-independent core provides the device pipeline with many services. For example, the XGL core can perform generic error checking, backing store, and deferral mode handling. The core also keeps track of XGL Context state and provides interfaces that allow the device pipeline to get information on attribute settings. In addition, the XGL core provides the device pipeline with:

- A quick test to determine whether any view model or coordinate system attributes have changed.
- Utilities that the device pipeline can use in rendering.

A simple view of the XGL core and a device pipeline that has implemented a complete LI-1 loadable interface for the API primitive `xgl_multipolyline()` looks like Figure 2-2 on page 13.



*Figure 2-2* High-Level View of the XGL Primitive Call Processing

For more information on the XGL architecture and for illustrations on the architecture of the device pipeline, see the *XGL Architecture Guide*.

## *Porting Task*

During the initial design phase for a device pipeline, you must choose the primary interface level (either LI-1, LI-2, or LI-3) for the port. This section presents some guidelines for choosing an interface level and, as an example, provides a brief overview of the steps involved in porting at the LI-1 level.

### *Choosing a Loadable Interface Level*

An important decision when you begin your port is to determine which loadable interface level to begin implementing first. Depending on your goals and your hardware, you may want to begin with LI-1 functions, LI-2 functions, or LI-3 functions. You can also focus on either 2D rendering or 3D rendering, as these are different paths. In some cases, the hardware will determine the loadable interface level that you port to, as follows:

- Consider an LI-1 port if your hardware provides a high level of graphics rendering capability, such as transforms, clipping, lighting, or accelerated scan conversion. Points are input to an LI-1 pipeline in model coordinates, and it is the device pipeline's responsibility to perform all rendering operations, including transforming the point data to device coordinates.
- Plan on an LI-2 port if your hardware is capable of rendering device coordinate primitives but is not capable of performing higher level operations such as depth cueing, transformations, lighting, or clipping. The LI-2 layer is provided for devices that can draw primitives if the device coordinates and color of the object are given and no further processing is required.
- Port to LI-3 if your device is a frame buffer that provides pixel-based operations and that does not provide graphics acceleration. The input to LI-3 is pixel data, and the frame buffer renders in device coordinates.

If you are writing a pipeline for a high-level graphics device, you may begin by implementing the basic put-pixel and get-pixel interfaces at the LI-3 level or by implementing one or more accelerated pipelines at the LI-1 level. There is no particular layer that you must begin with, but there are some performance trade-offs that you may want to consider.



### *Starting With an LI-3 Level Port*

A good way to begin, even for an LI-1 port, might be to start work on the LI-3 level using the LI-3 utility object RefDpCtx (Reference Device Pipeline Context). To implement the LI-3 layer with this object, you simply write functions to store the value of a pixel (set-pixel) and to retrieve the value of a pixel (get-pixel). Then, you can call the LI-3 interfaces using the RefDpCtx utility. XGL will only use your LI-3 device pipeline port at the end of a rendering operation. The XGL software pipeline will handle all other operations required for rendering.

Using RefDpCtx to implement LI-3 is the simplest, quickest route for porting XGL to your hardware. With the LI-3 level implemented in this way, you can begin working on window system interactions with DGA and on verifying your port using the Denizen test suite. (See the *XGL Test Suite User's Guide* for information on the Denizen test suite.)

Porting to the LI-3 level provides breadth of functionality rather than performance. This is the approach to take if your primary goal is to port XGL quickly to see your device running an XGL application. An LI-3 port is advantageous during the early states of implementing a device pipeline because it produces full XGL functionality with a minimal amount of effort by the porting team. Then, to improve performance, you can concentrate on the primitives that you decide are most important and rewrite their implementation at the LI-1 or LI-2 interface level.

### *Starting with an LI-1 Level Port*

An alternate approach is to focus on accelerated rendering and begin with LI-1 primitives. For example, if your hardware is designed to render triangles at high speed, it will be more advantageous to implement triangle renderers and the LI-1 triangle primitives than to implement a pixel interface at LI-3.

Writing a set of LI-1 level interfaces is not a simple task and can require significant time and resources. If you are new to programming in C++, it will take even longer. Optimizing the code for maximum performance will require even more development time. One way to organize work at the LI-1 level is to focus on a single area of acceleration, for example polylines, and implement the LI-1 level primitive for that area. With this approach, you can identify

design problems early. Once the LI-1 primitive is performing well, you can implement more LI-1 primitives using the design that you have developed for the first primitive.

## *Starting with an LI-2 Level Port*

If you are writing loadable interfaces for a device that renders in device coordinates only, you will implement LI-2 and LI-3 level interfaces and will not need to implement interfaces at the LI-1 level. In this case, you can choose whether to begin with the LI-2 layer or the LI-3 layer. As mentioned above, implementing LI-3 through RefDpCtx provides complete functionality in a relatively short time.

## *A Quick Look at Implementing an LI-1 Primitive*

Implementing an LI-1 level device handler is a large project consisting of several general steps. These steps are summarized in this section. While this section may make the task of writing an LI-1 level port seem simpler than it actually is, it is meant to help you divide the porting task into manageable subtasks or concepts. Each step includes references to later chapters that include the information needed to complete the task.

### ▼ **Decide which XGL primitives and attributes your hardware can accelerate.**

To determine which of the primitives and attributes your hardware can accelerate, consider the capabilities of your hardware and examine the scope of XGL functionality in the *XGL Reference Manual* and in Chapter 9, “Writing Loadable Interfaces”, of this book. Most likely you cannot implement all the XGL functionality on your device. Instead, you may want to focus on implementing only those features that your hardware can accelerate.

For those primitive-attribute combinations that your pipeline cannot handle, you can call the software pipeline for processing. To decide which primitives to implement in your pipeline, consider the kind of applications you are targeting with your device and the features that should be accelerated for those applications. Early identification of what to implement in your device pipeline will facilitate the process of porting XGL to your device.

▼ Write the `xgl_create_PipeLib()` routine.

Each pipeline must include a routine that creates an instance of the XGL device pipeline library object corresponding to the pipeline. This routine.

`xgl_create_PipeLib()`, is called through `dlsym()` after the device pipeline is dynamically loaded. See Chapter 3, “Pipeline Framework”, for information on this routine and for information on naming your pipeline so that XGL device initialization functions can load the pipeline at runtime.

▼ Subclass the set of classes that provide the device pipeline framework.

XGL provides a set of classes that, when initialized, provide a framework linking the device pipeline to the XGL core. You need to subclass from these classes for your device. Briefly, the XGL-provided classes are:

- `XglDpLib` – Maps to the shared library for your device.
- `XglDpMgr` – Maintains information about the physical device. You may want to put your device initialization routines in this class.
- `XglDpDev` – Constitutes the device-dependent part of the XGL Device object.
- `XglDpCtx2d` and `XglDpCtx3d` – Constitute the device-dependent part of the XGL Context object. Contain the loadable interfaces that the device implements.

These classes have a number of functions that you are required to implement as well as optional functions, such as the LI-1 and LI-2 loadable interfaces. For a summary of the required and optional functions in the device pipeline classes, see page 68. For detailed information on creating the device pipeline classes and objects, see Chapter 3, “Pipeline Framework”. For information and useful illustrations on the architecture of the device pipeline, see the *XGL Architecture Guide*.

You also need to consider your approach to implementing DGA. When you have implemented DGA and the skeleton for the XGL device pipeline classes, you will be able to create an X window and open an XGL Device on it.

### ▼ Choose a simple LI-1 primitive and implement geometry processing.

Once a window is available to render to, you can choose a primitive, such as `xgl_multipolyline()` to implement. The goal for this step is to render a simple piece of geometry, such as a line, on your hardware. To do this, you need to process the geometry data, converting it to a format appropriate for your hardware. You may also need to work out a way to initialize your hardware for each primitive.

Note that some window information, in particular the window clip list, is critical data. This means that it cannot be modified by another process while XGL is using it. Thus, the device pipeline must lock critical window data structures before rendering and unlock them when rendering is complete. This prevents the server from making changes to these data structures while an XGL rendering operation is taking place. See Chapter 8, “Window System Interactions” for more information on XGL’s interface to the window system.

Once you have succeeded in rendering geometry on your device, you have completed the important milestone of getting XGL to communicate with your hardware.

### ▼ Determine how to handle attribute processing.

Each XGL primitive has a set of attributes that affect it. The set of attributes for each primitive is noted in Chapter 9, “Writing Loadable Interfaces”. Your pipeline can get the attribute settings that it needs from the Context object and process the attribute changes using your pipeline `objectSet()` function.

Refer to Chapter 5, “Handling Changes to Object State” for information on design issues to consider as you implement attribute handling in your pipeline. When handling attribute changes, be aware that techniques that work for a simple primitive, such as multipolyline, may not work for more complex primitives, such as surface primitives.

At this time, you will also want to consider how to handle view model changes and coordinate system changes. XGL provides the view model derived data facility to assist you in implementing view model operations. Using derived data, you can set up objects that track the derived items important to your pipeline. See Chapter 7, “View Model Derived Data” for information on the processing of viewing and coordinate system changes.

Note that you may have to map the XGL attributes to attributes specific to your hardware so that the appropriate rendering occurs. Once you have determined what attributes you need to handle and how to handle them, you should think about how to structure the pipeline for performance. How you do this will depend on how your hardware saves Context state values. If you determine that your device cannot handle the current attribute setting for a primitive, you can fall back to the software pipeline for rendering.

▼ **Implement a design for falling back to the software pipeline.**

At each rendering call, the device pipeline must determine whether it can proceed. If it cannot, it can pass control to the software pipeline. See page 20 for information on falling back to the software pipeline.

Your pipeline must also manage state changes that may result when the application changes the Context it is using to render. Chapter 5, “Handling Changes to Object State” provides a brief discussion on context switching and hardware state updating. This chapter also provides information on handling the updating of state when the pipeline switches between interface layers. There are several pitfalls that you may encounter when switching loadable interface layers. Solving these design problems early in the porting process will simplify your overall task.

When you reach this point, you have worked through most of the porting process for a geometry operator. You should be familiar with problems that you need to resolve. At this point, you can look into implementing other types of functions, including functions that XGL does not provide, such as the `xgl_context_new_frame()` operator.

▼ **Implement `xgl_context_new_frame()` and a raster operator.**

There is a small subset of device-dependent operators that XGL does not implement in the software pipeline. The `xgl_context_new_frame()` operator is one of these operators. The new frame operator clears the screen and may be required each time rendering occurs. You may want to implement `xgl_context_new_frame()` early in your development schedule.

The next step might be to implement a pixel or raster operator, such as `xgl_context_copy_buffer()`, which is another operator that the device pipeline must provide. Implementing a pixel operator after a geometry primitive will help you understand the range of possible functions that you

must handle. When you have implemented a complete geometry primitive and a pixel operator, you have a good idea of the complete task of writing an LI-1 device pipeline.

## *Testing Your Implementation*

To verify that your device pipeline produces images that conform to XGL's reference images, run the Denizen Test Suite, which is supplied with the XGL DDK. The Denizen Test Suite is a group of shell scripts and C programs designed to use the XGL library to render objects and evaluate results. Denizen contains approximately 580 test programs that test every XGL function and the major internal components of the XGL library.

Your device handler should produce Denizen pass rates similar to those measured for Sun's reference frame buffers (8- and 24-bit nonaccelerated frame buffers). The Denizen Test Suite is not intended to be a debugging tool, but it is intended to provide a verification tool so that you can ensure the accuracy of your implementation. For information on using the Denizen Test Suite, see the *XGL Test Suite User's Guide*.

## *Calling the Software Pipeline*

When the device pipeline is called, if it can render the geometry, in most cases it will take control and render to the hardware at that point. If the device pipeline cannot perform the LI-1 or LI-2 processing, the device pipeline must call the software pipeline to process the primitive.

The software pipeline may also call the device pipeline. For example, if your device pipeline has not implemented a stroke text primitive, it can call the software pipeline LI-1 stroke text function. The software pipeline will tessellate the text into lines and then call the device pipeline multipolyline function to render the lines.

## *Device Pipeline Options to Rendering Calls*

In response to a rendering call, the device pipeline has several options:

- The device pipeline can fully render the primitive.

- The device pipeline can select its loadable interface function with the `opsVec[ ]` function array. The `opsVec[ ]` array is a dynamic array of loadable interface function pointers. It is the device pipeline's responsibility to set the entries in the `opsVec[ ]` array to point to the appropriate functions. For more information about the `opsVec[ ]` array, refer to "Defining the Device Pipeline-Context Class" on page 43 and the *XGL Architecture Guide*.
- The device pipeline can call the software pipeline for components of primitives that the hardware cannot render. For example, the software pipeline can render a subset of the primitive data, as in the case of a polygon that the device pipeline cannot handle in a `xgl_multi_simple_polygon()` call.
- The device pipeline can fall back on the software pipeline for the entire primitive as in the case of a clipped polygon that the device pipeline cannot handle in a `xgl_polygon()` call.

## *What Else You Should Know*

### *Accessing External Files at Runtime*

The XGL system may require a number of external files during the execution of an XGL application. For example, the device pipelines are dynamically loaded shared object files that must exist in a directory tree in a location known to XGL so that XGL can load them. The XGL library also requires external files for the software pipeline, error messages, stroke fonts, and configuration information. These external files exist within the directory tree that is created when the XGL files are installed. The top of this directory tree is pointed to by the `XGLHOME` environment variable. The value of `XGLHOME` is used internally by XGL when it searches for any of the external files.

To retrieve the value of `XGLHOME` from the XGL core, use the static function `XglGlobalState::getXglHome()` as shown below.

```
const char*          xgl_home;  
xgl_home = XglGlobalState::getXglHome();
```

## Directory Structure for the XGL DDK

Figure 2-3 illustrates the current XGL DDK directory structure. The XGL DDK package includes sample source code for the XGL reference loadable device pipelines.

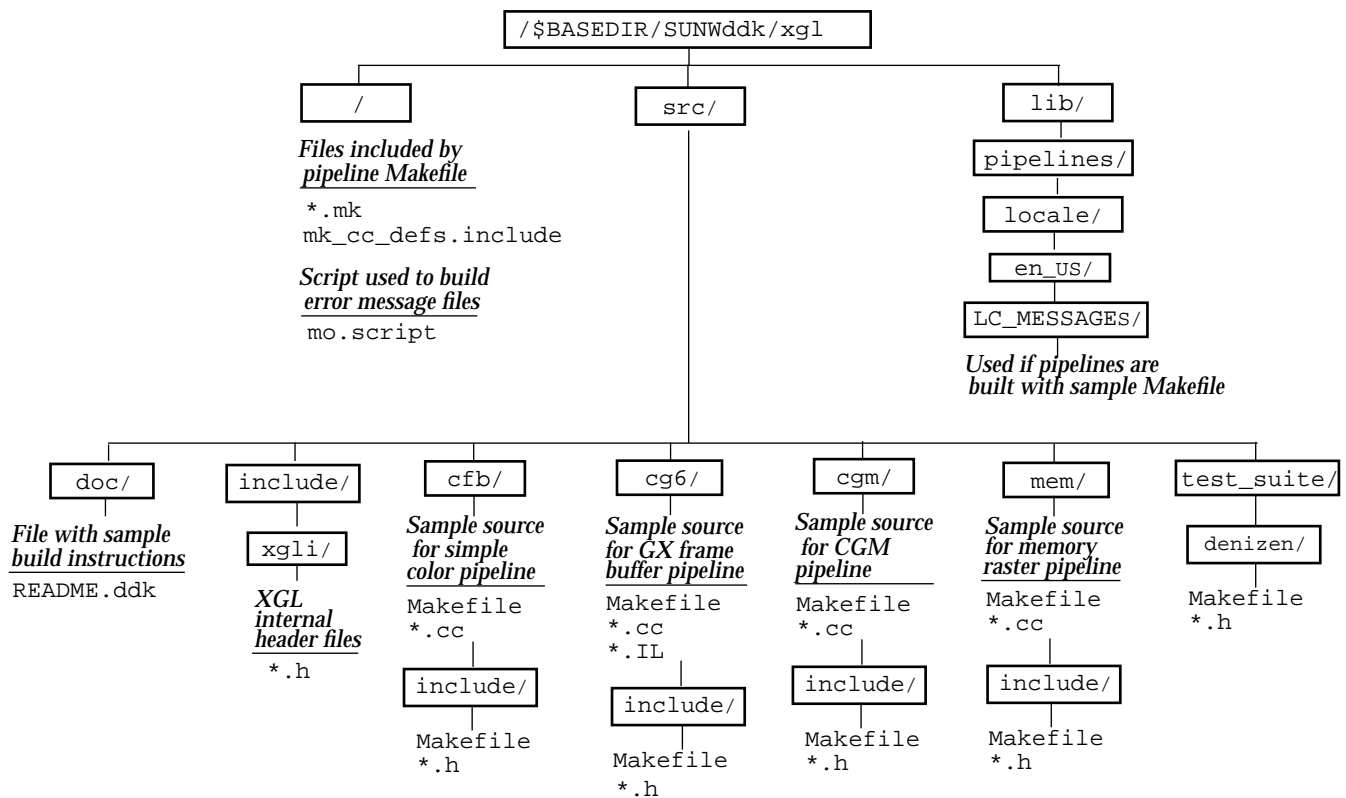


Figure 2-3 XGL DDK Directory Structure



## *Error Reporting for XGL Device Pipelines*

XGL provides an error-reporting mechanism that is used when an error is detected during the execution of an XGL application. In order for an error to be reported to the application, you must explicitly add code to handle error conditions. The easiest way to call the error notification function is with the use of error-reporting macros that are described in the following section.

### *Error Reporting Macros*

The recommended method to call the XGL error-handling function from within the pipeline code is to use one of two error-reporting macros that are defined in the file `<xgl_dirs>/src/include/xgli/ErrorMacros.h`. The macros are `XGLI_ERROR` and `XGLI_DI_ERROR`. Their interfaces are defined as follows:

<code>XGLI_ERROR(sys_state, type, category, error_id, object, op1, op2)</code>		
<code>XglSysState*</code>	<code>sys_state</code>	Pointer to current system state; can be NULL; if NULL, then internal function will first get system state pointer from global state
<code>Xgl_error_type</code>	<code>type</code>	Error type for the particular error
<code>Xgl_error_category</code>	<code>category</code>	Error category for the error
<code>char*</code>	<code>error_id</code>	Identification string for the error
<code>Xgl_obj_type</code>	<code>object</code>	Object type of currently active object
<code>char*</code>	<code>op1</code>	Optional operand for this error
<code>char*</code>	<code>op2</code>	Optional operand for this error

```
XGLI_DI_ERROR(sys_state, error_id, object, op1, op2)
XglSysState* sys_state Pointer to current system state;
                        can be NULL; if NULL, then internal
                        function will first get system
                        state pointer from global state
char* error_id Identification string for this
                        error
Xgl_obj_type object Object type of currently
                        active object
char* op1 Optional operand for this error
char* op2 Optional operand for this error
```

The default error notification function prints internationalized error messages that are retrieved from error message files stored in the directory `{path}/{LANG}/LC_MESSAGES/filename.mo`, where `{path}` is `$XGLHOME/lib/locale` if `$XGLHOME` is set, or `/opt/SUNWits/Graphics-sw/xgl/lib/locale` if `$XGLHOME` is not set. More than one error message file may exist in this directory.

The error message files are binary encoded. The file `xgl.mo` contains error messages for errors that could occur in either the device-independent XGL code (`libxgl.so`) or the device-dependent XGL code (the pipelines). The other `*.mo` files, named `xgl<company abbrev><pipeline abbrev>.mo`, contain error messages for errors that can only occur within a specific pipeline.

The `XGLI_ERROR` macro can be used to call the error-reporting function for errors that are defined in either `xgl.mo` or in the pipeline `*.mo` files. `XGLI_DI_ERROR`, however, is used only to report errors defined in `xgl.mo`.

The specific error message used by the error-handling function is identified by the `error_id` parameter passed to these macros. The `error_id` is a character string of one of the following forms, where `##` is the error number specified in the error message file:

1. `di-##` - For error messages from the `xgl.mo` (device-independent) error file
2. `<pipeline abbrev>-##` - For error messages from pipeline `.mo` files associated with the originally supported SunSoft/SMCC frame buffers

3. `xgl<company abbrev><pipeline abbrev>-##-` For error messages from independent hardware vendor (IHV) pipeline .mo files

In order to determine what error messages exist in the error files, English clear-text (ASCII) versions of the files are located in the following directories:

- For `xgl.mo` - `{path}/include/xgl/xgl_errors_di.po`
- For pipeline error message files -  
`{path}/src/<pipeline>/include/xgl_errors_<pipeline>.po`

The \*.po files are of the form:

```
msgid"Key String" (same as the error_id string)
msgstr"Translatable error message string"
```

The UNIX utility `msgfmt` encodes the \*.po ASCII files to create the \*.mo binary-encoded versions, which must be placed in the `locale` directory described above.

Other parameters passed to the error macros are self-explanatory. For more information on error types and categories, see "Error Handling" in the *XGL Architecture Guide*. The operand values may be used to add useful noninternationalized information (such as numbers or XGL attribute names) to the error report.

---

**Note** – The macros `XGLI_ERROR` and `XGLI_DI_ERROR` use the current operator set by the XGL core wrappers during error reporting. A device pipeline should never set the current operator in the pipeline.

---

### *Example of Error Reporting Using the Error Macros*

Suppose you want to check for a `malloc` error in your pipeline code. The following steps describe how this is done.

1. Search the ASCII clear-text version of the device-independent and pipeline error files for an error message corresponding to the error condition for which you are checking. In this case, the following error message is defined in `xgl_errors_di.po`:

```
msgid    "di-1"
msgstr   "malloc or new failed: out of memory"
```

2. Add the following `#include` to your source code module:

```
#include "xgli/SysState.h"
```

3. Add a call to one of the two error-reporting macros where you detect the error in your code:

```
if (!(pts = (Foo *)malloc(bar * sizeof(Foo)))) {
    XGLI_DI_ERROR (system_state, "di-1", XGL_3D_CTX, NULL, NULL);
    return (-1);
}
```

If the handle to the System State object is not known, you can call the macro using a `NULL` value for the System State parameter as shown below:

```
XGLI_DI_ERROR ((XglSysState *)NULL, "di-1", XGL_3D_CTX,
               NULL, NULL);
```

The error-handling function gets the error file, finds the proper error message string corresponding to the *error\_id* passed by the user, assigns values to internal error attributes, and calls an error notification function (either the default or one set by the XGL application). The default error notification function prints an error message to `stderr`. For example, in the case of the `malloc` error above, the following message is printed:

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

## *Creating a Pipeline Error Message File*

As described in the previous section, two types of error message files are delivered with XGL. You can create a new error message file for your pipeline and add error messages to it. Error messages in this file must be specific to the pipeline and should not duplicate error messages that are already available in the device-independent error message file.

Follow these steps to create a new error message file:

1. Use the template named `xgl_errors_template.po` in `$XGLHOME/src/sample_dp/include`.
2. Change all occurrences of `<company abbrev>` and `<pipeline abbrev>` so that they correspond to your company abbreviation and pipeline (device) abbreviation.
3. Add error messages at the end of the file. Two lines are required for each error message: a `msgid` line and a `msgstr` line. See the description in the template file or examine the `xgl_errors_di.po` file for more information.
4. Add the following lines to your Makefile in the directory where the clear-text version (the `.po` file) of the error message file is located:


```
xgl<company abbrev><pipeline abbrev>.mo: xgl_errors_<pipeline abbrev>.po
msgfmt xgl_errors_<pipeline abbrev>.po
```

You may add new error messages to the end of the error file once it has been created. Then, use the `XGLI_ERROR` macro described above to call the error handler with the error messages you define in your `.po` file.



## Pipeline Framework

---

3 

This chapter presents information on the classes and objects that connect XGL device-independent code with the device pipeline. The following topics are covered:

- Creating the required device pipeline classes
- Providing renderers optimized for performance-critical primitives
- Description of required and optional device-dependent functions
- Pipeline naming conventions and versioning
- Using the XGL core for backing store support

Note that in XGL the term *device* refers to both the physical hardware device and the XGL API Device object. The API Device object is an abstraction of the graphics display device. Internally, it consists of two objects: a device-independent object and a device-dependent object. For more information on the internal components of the API Device object, see the *XGL Architecture Guide*.



As you read this chapter, you will find it helpful to have access to the header files for the device pipeline classes. These files are:

- PipeLib.h and DpLib.h
- DpMgr.h
- DpDev.h, DpDevRaster.h, DpDevWinRas.h, and DpDevMemRas.h
- DpCtx2d.h and DpCtx3d.h

## Overview of the Pipeline Framework

The XGL architecture has a device-independent component (XGL core) and a device-dependent component. The device-dependent component consists of interfaces to the pipeline code. Because the device-independent component of XGL must interact smoothly with the device pipeline, XGL provides a set of classes that, when subclassed by the device pipeline, creates a *framework* that allows XGL to pass information to and from the device pipeline. Setting up the basic pipeline framework is one of the primary tasks in writing a device pipeline.

A pipeline implementation needs to derive classes from four different class hierarchies. This means that for your pipeline, you must subclass at least one pipeline class from each of the pipeline class hierarchies. Objects instantiated from the pipeline subclasses provide the functionality that the XGL device-independent code requires. The four pipeline class hierarchies are:

- Device pipeline library (DpLib)
- Device pipeline manager (DpMgr)
- Device pipeline device (DpDev)
- Pipeline-context (DpCtx2d and DpCtx3d)

At a minimum, your implementation must define five derived classes (one each from the device pipeline library, the device pipeline manager, and the device pipeline device hierarchies, and two from the device pipeline-context hierarchy) that form the basic framework of a device pipeline. Figure 3-1 on page 31 shows the XGL-supplied class header files, header files subclassed by the pipeline implementation, and the objects that are instantiated.

Each of the device pipeline derived classes contains functions that you must implement. In some cases, the functions simply create the next level of the hierarchy; in other cases, there are API-level functions or attributes that you must support. Several classes also include optional functions for operations that depend on the hardware.



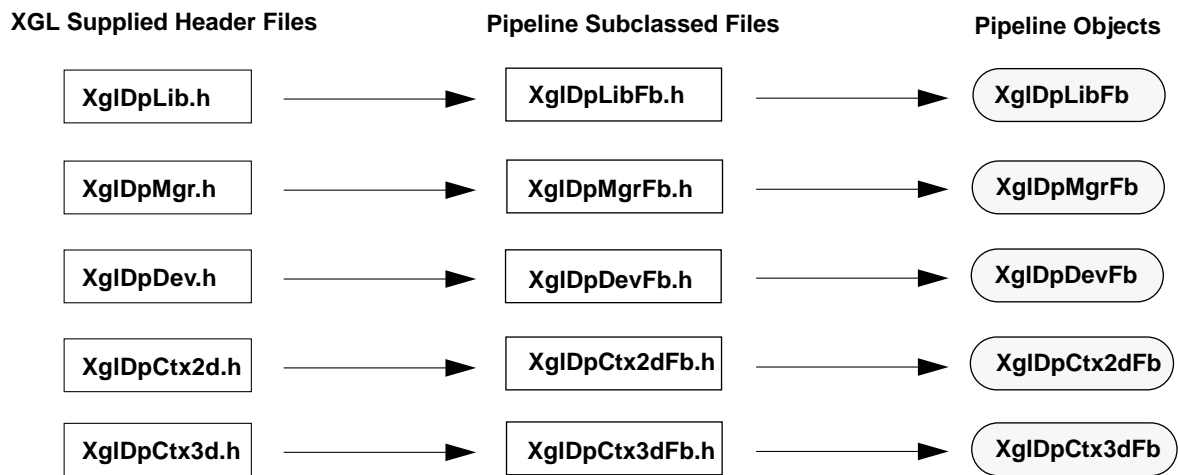


Figure 3-1 Device Pipeline Framework Classes

In addition to providing the required classes and functionality, you must include in your library a function called `xgl_create_PipeLib()`, which creates an object that represents the pipeline library. You must also name your pipeline appropriately so that XGL can load the pipeline object.

To summarize, this is what you must do to set up the framework for your pipeline:

1. Define the `xgl_create_PipeLib()` routine.
2. Define an `XglDpLib` class for your pipeline and implement the required functions.
3. Define an `XglDpMgr` class for your pipeline and implement the required functions.
4. Define an `XglDpDev` class for your pipeline and implement the required functions. Implement any of the optional functions that you need for your hardware.
5. Define two `XglDpCtx` classes for your pipeline, one for 3D and another for 2D. These classes contain an array of function pointers to primitives.
6. Name your pipeline according to the conventions noted on page 51.

These steps are discussed in the following sections. For information on how the XGL core instantiates the pipeline objects and loads the pipeline during device creation, and for illustrations showing how these classes are associated at runtime, see the *XGL Architecture Guide*.

Note that this chapter contains a number of source code examples. You can copy or modify these examples as long as the resulting code is used to create a loadable pipeline for XGL.

## Setting Up the Pipeline Framework

### Defining `xgl_create_PipeLib()`

As a first step in writing a device pipeline, you must write a routine that creates an instance of the XGL device pipeline object corresponding to your device pipeline. This routine is named `xgl_create_PipeLib()`, and it must be included with each pipeline. The routine is called by the XGL core through `dlsym()` (an interface routine in the Solaris dynamic linking mechanism) after the device pipeline is loaded. This function is declared as follows:

```
extern "C" XglPipeLib* xgl_create_PipeLib()
```

Below is a basic implementation of this function, where `XglDpLibSampDp` represents the name of the `XglPipeLib` subclass that the device pipeline creates.

```
XglPipeLib* xgl_create_PipeLib()
{
    return new XglDpLibSampDp;
}
```

Note that the extern "C" declaration is needed to disable the C++ name mangling on the function name.

## Defining the Device Pipeline Library Class

Next, you must subclass from the device pipeline library class hierarchy to create your device pipeline library (XglDpLib) class. An object from this class does the following:

- Represents a loaded device pipeline and maps to the .so shared object for that pipeline. For each pipeline that is loaded into the XGL environment, there is an XglDpLib object created by the pipeline function `xgl_create_PipeLib()`.
- Provides for the creation, management, and destruction of device pipeline manager objects and allows more than one device pipeline manager object to share hardware or software resources.
- Provides a location for you to place for data relevant to your pipeline library as a whole.

The base class of the device pipeline library hierarchy is XglPipeLib. This class derives to the more specific device pipeline library and software pipeline library classes XglDpLib and XglSwpLib. Your individual device pipeline implementation will derive from XglDpLib. See the files `PipeLib.h` and `DpLib.h` for the definition of these classes. A minimal definition of a pipeline library class is shown here. You can copy or modify the source code samples in this chapter as long as the resulting code is used to create a loadable pipeline for XGL.

```
#include "xgl/xgl.h"
#include "xgli/DpLib.h"
#include "DpMgrSampDp.h"

class XglDrawable;

extern "C" XglPipeLib* xgl_create_PipeLib();

class XglDpLibSampDp : public XglDpLib {
    friend XglPipeLib* xgl_create_PipeLib();
private:
    XglDpLibSampDp() { dpMgr = NULL; }
    ~XglDpLibSampDp();
    //
    // Device-pipelines Dependent Functions -
    // Redefine in Device Pipelines
```

```
virtual XglDpMgr* getDpMgr(Xgl_obj_type,
                          XglDrawable* drawable=NULL);

XglDpMgrSampDp* dpMgr; // I only have one dpMgr
};
```

Note that if there are two frame buffers on a system and both are of the same type, such as GX, there is one XglDpLib object. If the two frame buffers are different types, such as one GX and one IHV-provided frame buffer, there are two XglDpLib objects, one for each device pipeline. The Global State object in the XGL core keeps a list of XglDpLib objects so that it can destroy them when XGL is closed. See the *XGL Architecture Guide* for information on the Global State object.

**Note** – If the device pipeline needs to establish exclusive control of any device-dependent behavior for client applications, this control is handled by the device pipeline objects, as the XGL core does not handle device-specific control of applications. If the control is needed for all clients of the same type of frame buffer (regardless of the number of frame buffers), then the XglDpLib object should maintain the control. If the control is required for each frame buffer (if there is more than one), then the XglDpMgr object should handle the control.

## *XglDpLib Functions That You Must Override*

The XglDpLib class contains one virtual function that you must override for your pipeline implementation:

```
getDpMgr(Xgl_obj_type, XglDrawable* drawable)
```

This function is called by the XGL core's device creation routine when it creates a new XglDpMgr object. The *drawable* parameter enables the device pipeline to distinguish between different physical frame buffers of the same type. Note that this pointer is transient and should not be cached. The *Xgl\_obj\_type* parameter is currently ignored; in future releases, it may be used to allow a pipeline to create more than one type of Device object.

For implementations that handle multiple frame buffers, the XglDpLib object may keep a list of previously created XglDpMgr objects. In this case, the `getDpMgr()` function should check the list for an existing XglDpMgr object associated with the Device type and Drawable object and retrieve the XglDpMgr object if it exists.

Since the number of device pipeline manager objects your pipeline needs depends on the capabilities of your hardware, the creation, management, and destruction of `XglDpMgr` objects is left to your individual device pipeline implementation. Depending on the functions performed by an `XglDpMgr` object, one `XglDpMgr` can be created for each frame buffer, or one `XglDpMgr` can be created for all frame buffers of the same type. Typically, the device pipeline provides one `XglDpMgr` object for each frame buffer, but the pipeline can manage `XglDpMgr` objects in other ways as well. Note that the destruction of the `XglDpMgr` objects should be handled in the `XglDpLib` destructor function, which the XGL core invokes during `xgl_close()`.

Systems with multiple frame buffers can have frame buffers of either the same type or different types. If the frame buffers are of the same type, the `XglDpMgr` objects are created by the unique `XglDpLib` object for that pipeline. If the frame buffers are different types, each `XglDpMgr` object is created by the `XglDpLib` object corresponding to the device pipeline for that frame buffer.

For device pipelines that only need one `XglDpMgr`, such as a memory raster pipeline, the `getDpMgr()` function will return the same `XglDpMgr` object every time it is needed. A sample implementation of `getDpMgr()` is shown below.

```
XglDpMgr* XglDpLibSampDp::getDpMgr(Xgl_obj_type,
                                   XglDrawable* drawable)
{
    XglDpMgr*    dpMgr;

    dpMgr = dpMgrList.getDpMgr(drawable->getDevFd());
    if (dpMgr == NULL) {
        dpMgr = new XglDpMgrSampDp(drawable->getDevFd());
        dpMgrList.addDpMgr(dpMgr);
    }
    return dpMgr;
}
```

**Note** – Although you can initialize your hardware in any of the framework classes, a good place to initialize the hardware is in your `XglDpMgr` constructor, since this is where the frame buffer is first notified that XGL is going to use it.

## Defining the Device Pipeline Manager Class

The next step in setting up the pipeline framework is to define the device pipeline manager (XglDpMgr) class. An object from this class does the following:

- Provides for the creation of the device pipeline device objects. This class allows multiple device pipeline device objects to share the physical resources of a device.
- Maintains information about the physical hardware device. If there are multiple frame buffers of the same type on a system, there are either multiple XglDpMgr objects, or the XglDpLib object handles the multiplicity in some other way. If the frame buffers are of different types, for example, one GX and one Cg3 (color frame buffer), there are two XglDpMgr objects: one XglDpMgrGx object and one XglDpMgrCfb object.

See the file `DpMgr.h` for the definition of this class. A minimal definition of a device pipeline manager class is shown here as `XglDpMgrSampDp`.

```
class XglDevice;
class XglDpDev;
class XglDrawable;

class XglDpMgrSampDp : public XglDbgObject {
public:
    virtual ~XglDpMgr();
private:
    //
    // Device-pipelines Functions - Redefine in Device Pipelines
    //
    virtual XglDpDev* createDpDev(XglDevice*,
                                  Xgl_obj_desc* bkstore_desc = NULL);
    virtual void inquire(XglDrawable*, Xgl_inquire*);
};
```

To limit the number of XglDpMgr objects, you can deny the creation of new XglDpMgr objects by returning `NULL` from `XglDpLibYourFb::getDpMgr()`. You can also limit the creation of a new XglDpDev object by returning `NULL` from `XglDpMgrYourFb::createDpDev()`. Recoverable errors from the XGL core result in those situations.

## *XglDpMgr Functions That You Must Override*

XglDpMgr includes the following virtual functions that you must override:

`XglDpDev* createDpDev(XglDevice*, Xgl_obj_desc*)`

Invokes the creation of the device-dependent part of the XGL Device object. The *XglDevice* argument is cast to a pointer to the type of Device being created, such as *XglRasterWin* or *XglRasterMem*. The *Xgl\_obj\_desc* argument is a pointer to a structure containing additional information about the XGL Device object. The XGL core uses the information in this structure, and the device pipeline normally does not need it. However, when backing store is enabled, this argument provides information about the parent device that the backing-store device can use or ignore.

A sample implementation of the `createDpDev()` function to create a window raster device is shown below. Note that:

- *XglDpMgrSampDp* represents the name of the device type, for example *XglDpMgrGX*.
- *XglDpDevSampDp* represents the name of the *XglDpDev* subclass that this device pipeline creates, for example *XglDpDevGx*.

```
XglDpDev* XglDpMgrSampDp::createDpDev (XglDevice* device,
                                         Xgl_obj_desc*)
{
    return new XglDpDevSampDp(this, (XglRasterWin*)device);
}
```

`void inquire(XglDrawable*, Xgl_inquire*)`

An API-level function that returns information on the acceleration features of your device.

For an implementation of the `inquire()` function, see the sample LI-1 port provided as part of this product. Note that `inquire()` might be called before the *XglDpDev* object is created. The `inquire()` function should use its *XglDrawable\** parameter to fill the contents of the *Xgl\_inquire* structure whose address is passed. Note also that the *XglDrawable* pointer passed into the `inquire()` function is transient and is destroyed afterward. For more information on the `inquire()` function, see the `xgl_inquire()` reference page in the *XGL Reference Manual*.

## Defining the Device Pipeline Device Class

Next, you must subclass from the device pipeline device hierarchy. An object from this hierarchy contains the device-dependent elements of an XGL Device object and is linked to the device-independent part of the Device object. An object instantiated from the `XglDpDev` class does the following:

- Creates the device pipeline-context objects
- Provides a device pipeline with the opportunity to exchange device information with XGL core via `get()` and `set()` functions
- Provides a storage location for data relevant to the window

In the case of a single XGL application with multiple windows, each `XglDpDev` object maps to a single window on the screen. If the application has multiple windows using the same underlying frame buffer, the `XglDpMgr` object for that frame buffer creates all the `XglDpDev` objects that the application needs. If the application runs on a system with more than one physical frame buffer, and the application creates multiple windows on each frame buffer, each `XglDpDev` object is created by the `XglDpMgr` object that corresponds to the frame buffer.

Although the `XglDpMgr` object creates multiple `XglDpDev` objects, it is not designed to keep track of these objects. Instead, for each XGL API-level Device object that is created, a pointer to the `XglDpDev` object is returned to the device-independent `XglRasterWin` object, and a pointer to the `XglRasterWin` object is stored in the `XglSysState` object list of existing Device objects. For more information on how pipeline objects are instantiated, see the *XGL Architecture Guide*.

The base class of the device-dependent device hierarchy is `XglDpDev`, which derives to `XglDpDevRaster` and then to `XglDpDevWinRas`, `XglDpDevMemRas`, or `XglDpDevStream`. Depending on the type of the device you are porting, your device pipeline will subclass from either `XglDpDevWinRas` (for window rasters), `XglDpDevMemRas` (for memory rasters), or `XglDpDevStream` (for stream devices). See the header files `DpDev.h`, `DpDevRaster.h`, `DpDevWinRas.h`, `DpDevMemRas.h`, and `DpDevStream.h` for the device-dependent hierarchy. Sample code for a minimal definition of a device pipeline device class for a window raster is shown below.



```

class XglDpDevSampDp : public XglDpDevWinRas {
    friend XglDpMgrSampDp;
private:
    XglDpDevSampDp(XglDevice* device) : XglDpDevWinRas(device) {}
    //
    // Device-pipelines Dependent Functions -
    // Redefine in Device Pipelines
    //
    virtual XglDpCtx3d* createDpCtx(XglContext3d*);
    virtual XglDpCtx2d* createDpCtx(XglContext2d*);

    virtual int copyBuffer(
        XglContext3d*,    //3D Context associated with dst mem_ras
        Xgl_bounds_i2d*,  //Rectangle
        Xgl_pt_i2d*);     //Position

    virtual int copyBuffer(
        XglContext2d*,    //2D Context associated with dst mem_ras
        Xgl_bounds_i2d*,  //Rectangle
        Xgl_pt_i2d*);     //Position
};

```

**Note** – When XglDpDevWinRas is created, a device pipeline should call XglRasterWin:setDgaCmapPutFunc() to register the callback function that updates the hardware color map. See page 135 for information on setDgaCmapPutFunc().

### *XglDpDev Functions That You Must Override*

A minimal implementation of the XglDpDev class includes several functions that you must override:

```

virtual XglDpCtx{2/3}d*
createDpCtx(XglContext{2/3}d*)

```

A virtual function to create the XglDpCtx objects. Two of these functions must be created, one for 2D and one for 3D. These functions must be implemented in the pipeline, or an error is returned.

```
virtual int
copyBuffer(XglContext{2/3}d*,Xgl_bounds_i2d*,Xgl_pt_i2d*)
```

Two virtual functions, one for 2D and one for 3D, where the destination device is the memory raster associated with the Context parameter and the source device is the pipeline XglDpDev object.

A minimal implementation of `createDpCtx()` that instantiates a pipeline-context object is shown below. In this example, `XglDpCtx3dSampDp` represents the name of the `XglDpCtx3d` subclass that the device pipeline creates.

```
XglDpCtx3d* XglDpDevSampDp::createDpCtx(XglContext3d* context)
{
    return new XglDpCtx3dSampDp(context);
}
```

## *XglDpDev Optionally Overridable Functions*

The `XglDpDev` class and its subclasses include a number of functions that you can override to perform operations specific to your device. These function declarations, along with their default actions, are defined in the header files for the `XglDpDev` classes and are listed in Table 3-1 on page 41. If the default behavior of your hardware matches the defaults that XGL has defined for these functions, it is not necessary to override these functions.

Note that this may be an incomplete list; for the most current list, check the header files. Information on default and return values is also available in the header files. See the descriptions beginning on page 52 for more detail on these functions.

Table 3-1 Device-Dependent Overridable Functions

Class	Function Name
XglDpDev	Xgl_vdc_orientation getDcOrientation() float getMaxZ() float getGammaValue()
XglDpDevRaster	void setRectList(const Xgl_irect[]) void setRectNum(Xgl_usgn32) void setSourceBuffer(Xgl_buffer_sel) void setSwZBuffer(XglPixRectMem*) void setSwAccumBuffer(XglPixRectMem*) void syncRtnDevice(XglRasterWin*)
XglDpDevWinRas	Xgl_accum_depth getAccumBufferDepth() Xgl_usgn32 getDepth() Xgl_color_type getRealColorType() XglPixRectMem* getSwZBuffer() XglPixRectMem* getSwAccumBuffer() Xgl_boolean needRtnDevice() void resize() void setBackingStore(Xgl_boolean) void setBufDisplay(Xgl_usgn32) void setBufDraw(Xgl_usgn32) void setBufMinDelay(Xgl_usgn32) Xgl_usgn32 setBuffersRequested(Xgl_usgn32) void setCmap(XglCmap*) void setPixelMapping(const Xgl_usgn32) void setStereoMode(Xgl_stereo_mode)
XglDpDevMemRas	void setCmap(XglCmap*) void setImageBufferAddr(Xgl_usgn32*) void setZBufferAddr(Xgl_usgn32*) void setLineBytes(Xgl_usgn32) Xgl_accum_depth getAccumBufferDepth() XglPixRectMem* getAccumBufferPixRect() XglPixRectMem* getImageBufferPixRect() XglPixRectMem* getZBufferPixRect()

## *Device Object Initialization*

It is important to be aware that XGL's API-level Device object consists of two internal objects: the device-dependent device object that is created by the device pipeline `XglDpMgr` object, and a device-independent object, such as `XglRasterWin`, that is created by the XGL core System State object. These two internal Device objects are linked by a pointer from the device-dependent object to the device-independent object. The API Device object was designed with separate device-independent and device-dependent components to isolate the device-dependent operations. This design allows you to define specific operations for your device.

When the XGL core asks the device pipeline to create an `XglDpDev` object, it passes a handle to the device-independent Device object:

```
XglDpDev* XglDpMgrYourFb::createDpDev(XglDevice* device)
```

At creation time, the `XglDpDev` object gets from the `XglDevice` object all the information it needs about the device-independent attributes. The device-independent values are valid at this time. Most of the `set . . . ( )` functions are called later when the application changes device-dependent parameters through the API.

After getting the pointer to the `XglDpDev` object, the `XglRasterWin` object calls the device-dependent object's `get . . . ( )` functions (which you should override if the default behavior is incorrect for your device) to complete its own initialization. You should not expect these device-dependent attributes, which provide information such as the DC orientation or the device color type, to be meaningful during the `XglDpDev` object creation. Later in the process of pipeline initialization, `XglRasterWin` will call `set . . . ( )` functions that allow the `XglDpDev` object to complete its initialization with the correct device-dependent data.

## Defining the Device Pipeline-Context Class

The final step in creating your pipeline framework classes is to subclass from the device pipeline-context hierarchy. There is one device pipeline-context object per Device-Context pair. If your pipeline supports applications that render in 2D and 3D, then two subclasses are needed, one descending from `XglDpCtx2d` and the other from `XglDpCtx3d`.

The two `XglDpCtx` classes contain the interfaces for the 2D and 3D LI-1, LI-2, and LI-3 primitive layers. The LI-1 and LI-2 interfaces are member functions that you can override in your device pipeline if you choose. If the LI-1 and LI-2 primitives are not implemented by the pipeline, the device pipeline-context object automatically uses the software pipeline. Thus, for LI-1 and LI-2 primitives, the device pipeline does not need to override any functions that the hardware does not support. The LI-3 functions are defined in the `XglDpCtx` classes and must be implemented by the device pipeline.

An `XglDpCtx` class for a 3D pipeline in which only multipolylines are implemented might look like the following sample code.

```
class XglDpCtx3dSampDp : public XglDpCtx3d {
    friend XglDpDevSampDp;
private:
    XglDpCtx3dSampDp(XglContext3d* context) :
        XglDpCtx3d(context) {
        opsVec[XGLI_LI1_MULTIPOLYLINE] =
            XGLI_OPS(XglDpCtx3dSampDp::lilMultiPolyline);
        opsVec[XGLI_LI_OBJ_SET] =
            XGLI_OPS(XglDpCtx3dSampDp::objectSet);
        opsVec[XGLI_LI_MXG_RCV] =
            XGLI_OPS(XglDpCtx3dSampDp::messageReceive);
    }
    //
    // Device-pipeline Dependent Functions
    //
    void lilMultiPolyline(Xgl_bbox*, Xgl_usgn32, Xgl_pt_list*);

    // Function to handle ctx related changes
    void objectSet (const Xgl_attribute*);

    //Function to draw the line through the hardware
    void drawLine (Xgl_usgn32, Xgl_pt_list*);
};
```

The LI functions are described in Chapter 9, “Writing Loadable Interfaces”. For more information on switching between the device pipeline and the software pipeline, see page 20.

## *Rendering through the XglDpCtx Object*

When a primitive is called, the XGL core maps the API call to an internal C++ call in a wrapper function. The wrapper passes the primitive call directly to the device pipeline through the `opsVec[]` array. The `opsVec[]` is a dynamic array of function pointers to LI functions. It is defined by the XGL core and maintained by the device pipeline. The array is defined as shown below.

`XGLI_OPS` is defined as `(void(XglDpCtx::*)())` in `DpCtx.h`.

```
XglDpCtx3d::XglDpCtx3d(XglDevice*    dev,
                       XglContext3d* context) : XglPipeCtx3d(context)
{
    //
    // Initialize DI opsVec[]
    //
    opsVec[XGLI_LI1_NEW_FRAME] =
        XGLI_OPS(XglDpCtx3d::li1NewFrame);
    opsVec[XGLI_LI1_GET_PIXEL] =
        XGLI_OPS(XglDpCtx3d::li1GetPixel);
    opsVec[XGLI_LI1_SET_PIXEL] =
        XGLI_OPS(XglDpCtx3d::li1SetPixel);
    opsVec[XGLI_LI1_SET_MULTI_PIXEL] =
        XGLI_OPS(XglDpCtx3d::li1SetMultiPixel);
    opsVec[XGLI_LI1_SET_PIXEL_ROW] =
        XGLI_OPS(XglDpCtx3d::li1SetPixelRow);
    opsVec[XGLI_LI1_COPY_BUFFER] =
        XGLI_OPS(XglDpCtx3d::li1CopyBuffer);
    opsVec[XGLI_LI1_ACCUMULATE] =
        XGLI_OPS(XglDpCtx3d::li1Accumulate);
    opsVec[XGLI_LI1_CLEAR_ACCUMULATION] =
        XGLI_OPS(XglDpCtx3d::li1ClearAccumulation);
    :
    :
    :
}
```

When the application calls a primitive, the wrapper forwards the API function call to the device pipeline, as shown in the following sample wrapper function:

```
void xgl_multipolyline(Xgl_ctx      ctx,
                      Xgl_bbox*    bounding_box,
                      Xgl_usgn32    num_pt_lists,
                      Xgl_pt_list*  pl)
{
    XglDpCtx* dp = ((XglCtxObject*)ctx)->getDp(); //get dp pointer
    (dp->*( //call dp function pointed to by mpline array entry
        (void(XglDpCtx::*)(Xgl_bbox*,Xgl_usgn32,Xgl_pt_list*))
        (dp->currOpsVec[XGLI_LI1_MULTIPOLYLINE])
        )
    )(bounding_box,num_pt_lists,pl); // function called with API
                                   // args
}
```

**Note** – At LI-1, API geometry data is passed to the device pipeline unchanged.

### *Required Initialization of the opsVec[] Function Array*

The XGL core initializes the opsVec[ ] array to a set of default function pointers that point to the software pipeline LI-1 primitives. It is the device pipeline's responsibility to override the entries in the opsVec[ ] array to functions that the device pipeline has implemented. This can occur at initialization of the pipeline XglDpCtx object (when the Device is set on the Context) or during program execution. In deciding how to set up your pipeline's opsVec[ ] array, you have three cases to consider:

- Primitives that the pipeline does not implement
- Primitives that are not critical to performance
- Primitives that are critical to performance

Designing the opsVec[ ] array to handle these cases is discussed below.

## ***Using the Default Software Pipeline Renderer***

If your device pipeline has not implemented a particular LI-1 or LI-2 primitive, you can use the default `opsVec[]` array entry. The `opsVec[]` entries are loaded with calls to the software pipeline at LI-1 or LI-2 by default. The `XglDpCtx3d` default call to the software pipeline looks like this:

```
void XglDpCtx3d::lilMultiPolyline(Xgl_bbox*    bounding_box,
                                   Xgl_usgn32   num_pt_lists,
                                   Xgl_pt_list* pl)
{
    if (error_checking) {
        do_error_checking()
    }
    swp->lilMultiPolyline(bounding_box, num_pt_lists, pl);
}
```

In this case, you do not need to change the `opsVec[]` array. Your `XglDpCtx` object will inherit the default software pipeline calls.

## ***Implementing a Generic Renderer***

If your pipeline implements a primitive, but the primitive's performance is not critical, the pipeline can load a pointer to its primitive function when the Device is set on the Context and not reset it later. This function will be called whenever the application calls a primitive.

To provide a renderer, declare the function as a member of your pipeline `XglDpCtx`, and in the `XglDpCtx`'s constructor, put a pointer to the function in the appropriate entry of the `opsVec[]` array. A list of `opsVec[]` array indices can be found in `DpCtx.h`.

An example of initializing the `opsVec[]` array for a device pipeline LI-1 multipolyline is shown below:

```
//
// Install multipolyline
//
opsVec[XGLI_LI1_MULTIPOLYLINE] =
    XGLI_OPS(XglDpCtx3dExampleDp::lilMultiPolyline);
```



### ***Implementing a Performance-Critical Renderer***

If your device pipeline implements a primitive whose performance is critical, you may want to create a set of renderers for this primitive, including:

1. A single generic renderer that does error checking and handles point type changes, attribute changes, and transform changes.
2. A set of fast renderers that do not need to handle point type changes, etc. and that are tuned to specific combinations of attributes.

A generic renderer might look something like this:

```
//
// Get and return clip changed status.
//
// Result OR'ed and saved in "clipChanged" since
// drawable->clipChanged() does not retain clip status.
//
#define GX_CLIP_CHANGED(drawable) (clipChanged |= \
                                   (drawable)->clipChanged())
XglDpCtx3dGx::GenericMpline(Xgl_bbox*    bounding_box,
                             Xgl_usgn32  num_pt_lists,
                             Xgl_pt_list* pl)
{
    if (error_checking) {
        do_error_checking()
    }
    if (ctx != last_ctx) {
        // handle context change
    }
    if (pt_type != last_pt_type) {
        // handle point type change
    }
    if (prim != last_prim) {
        // handle primitive type change
    }
    if (GX_CLIP_CHANGED(drawable)) {
        // handle window type change
    }
    if (xforms_changed) {
        // handle transform changes
    }
    // Figure out which fast line renderer to use and set
    // opsVec[XGLI_LI1_MULTIPOLYLINE] to this line renderer.
```

```
// window locking
// model clipping

// Draw the multipolyline with the fast renderer you just set.
(this->*(void(XglDpCtx3dGx::*)
    (Xgl_bbox*, Xgl_usgn32, Xgl_pt_list*, Xgl_boolean)
    )(opsVec[XGLI_LI1_MULTIPOLYLINE])
    )(bounding_box, num_pt_lists, pl, FALSE);

// If nothing changes, this fast renderer will be
// called directly the next time.

// window unlocking
}
```

A fast renderer might look something like this:

```
XglDpCtx3dGx::FastMPLine()
{
    //state changes that require re-evaluation before rendering
    if (ctx!=last_ctx || pt_type!=last_pt_type ||
        prim!=last_prim || GX_CLIP_CHANGED(drawable)) {
        GenericMpline();
    }    return;
    // send the points to the hardware to render
}
```

A device pipeline rendering function can override opsVec[ ] entries at any time. You can design a renderer to support a particular set of attributes and install that renderer during program execution. This frees some renderers from having to test the attributes in each primitive call, and thus provides improved acceleration.

If a device decides to set an opsVec[ ] entry back to its default value, the opsVecDiDefault[ ] array can be used:

```
//
// Set opsVec[] back to default
//
opsVec[XGLI_LI1_MULTIPOLYLINE] =
    opsVecDiDefault[XGLI_LI1_MULTIPOLYLINE];
```

---

**Note** – In order for backing store to work correctly, the device pipeline must pass the parameter `gen_punt = FALSE` when it calls any `XglDpCtx` function through the `opsVec[]` array.

---

### *Calling the Software Pipeline*

If a device pipeline cannot accelerate the API arguments or the Context state (for example, API attributes, point type, or facet flags), the pipeline can call the software pipeline directly, as shown in this example:

```
void XglDpCtx3dSampDp:lilMultiPolyline(
    Xgl_bbox*      bounding_box,
    Xgl_usgn32     num_pt_lists,
    Xgl_pt_list*   pl)
{
    const XglStrokeGroup3d* cur_stroke = ctx->getCurrentStroke();

    // Check if I can render with the current attrs
    // e.g. If dp can handle only line style solid then
    // dp must call software pipeline
    if ( cur_stroke->getStyle() != XGL_LINE_SOLID) {
        swp->lilMultiPolyline(bounding_box, num_pt_lists, pl);
    }    return;

    // Draw the polyline
    drawLine(num_pt_lists, pl);
}
```

When a device pipeline can only render part of a primitive, it can fall back to the software pipeline for partial rendering of the primitive. For example, to handle a complex polygon in an `xgl_multi_simple_polygon()` call, the device pipeline can do the following:

```
//
// Second version of a fast renderer.
//
// Get and return clip changed status.
//
// Result OR'ed and saved in "clipChanged" since
// drawable->clipChanged() does not retain clip status.
//
```

```

#define GX_CLIP_CHANGED(drawable) (clipChanged != \
                                   (drawable)->clipChanged())

XglDpCtx3dGx::lilFastMsp(Xgl_facet_flags api_facet_flags,
                        Xgl_facet_list *api_facet_list,
                        Xgl_bbox *api_bbox,
                        Xgl_usgn32 api_num_pt_lists,
                        Xgl_pt_list *api_pt_list)
{
    //
    // API arguments that can't be handled by the device pipeline.
    //
    if (api_pt_list->pt_type & XGL__HOM) { // Homogeneous point
                                           // type
        swp->lilMultiSimplePolygon(api_facet_flags,
                                   api_facet_list,
                                   api_bbox, api_num_pt_lists,
                                   api_pt_list);

        return;
    }

    //
    // State changes that require re-evaluation before
    // rendering.
    //
    if (ctx != dp_saved_last_ctx_ptr ||
        api_pt_list->pt_type != dp_saved_last_pt_type ||
        api_facet_flags != dp_saved_last_facet_flags ||
        api_facet_list->facet_type != dp_saved_last_facet_type ||
        dp_saved_prim != dp_saved_last_prim ||
        GX_CLIP_CHANGED(drawable)) {

        lilMultiSimplePolygon(api_facet_flags, api_facet_list,
                               api_bbox, api_num_pt_lists, api_pt_list);

        return;
    }
    Xgl_pt_list *pl = api_pt_list;
    for(Xgl_usgn32 i = 0; i < api_num_pt_lists; pl++) { // for
                                                         each polygon
        if (api_facet_flags & XGL_FACET_FLAG_SHAPE_CONVEX)
            //
            // if convex
            // send the points to the hardware
            //
        else

```

```

//
// Parts of the primitive that can't be handled
//
swp->lilPolygon(api_facet_list->facet_type,
               api_facet_list->facets,
               api_bbox,
               1,
               pl);
    }
}

```

## Naming Your Device Pipeline

An XGL device pipeline must be named according to the following convention:

`xgl<COMPANY NAME><device name>.so.<major version>`

where:

- *<COMPANY NAME>* is a 4-letter capitalized abbreviation for the company that implements the device pipeline.
- *<device name>* is the abbreviated name of the device, which should be an abbreviated form of the name of the corresponding kernel device driver located in the `/dev` directory.
- *<major version>* is the major release number of the DDK associated with the particular release of XGL that is compatible with this device pipeline. The DDK major version number can be found in the header file `xgli/DdkVersion.h`.

For example, a Sun Microsystems Cg6 device pipeline with a major version number of 4 is named `xglSUNWcg6.so.4`. See page 63 for more information on version numbers.

The name of the pipeline is defined in the `Makefile` located in the device pipeline build area. The `Makefile` macro `LIB_NAME` must be set to the pipeline name.

When XGL attempts to load a pipeline, it issues a system call that returns the pipeline name, as defined above, for the active device. See page 58 for more information on how XGL loads a device pipeline.

## Optional Functions in Device-Dependent Classes

This section provides a brief description of the optional device-dependent operations provided in the `XglDpDev` class hierarchy. Many of these functions have a corresponding API attribute; in those cases, you can find more information about the attribute in the *XGL Reference Manual*.

### Overridable Functions in `DpDev.h`

```
virtual Xgl_vdc_orientation getDcOrientation()
```

Returns a value for the orientation of DC for the hardware device. The default value is `XGL_Y_DOWN_Z_AWAY`. Override this function if your device has a different orientation. This function is called by the XGL core as part of the Device object initialization.

```
virtual float getMaxZ()
```

Returns a value for the hardware device's maximum Z coordinate value. The default value is `XGLI_DEFAULT_MAX_DEPTH`, which is a constant defined as  $2^{24}-1$ . Override this function if your device has a different maximum Z value. This function is called by the XGL core as part of the Device object initialization.

```
virtual float getGammaValue()
```

The default implementation of this function returns a value of 2.22. The function also checks the environment variable `XGL_AA_GAMMA_VALUE` and returns the value that the environment variable is set to, if it is set. See Appendix B, "Software Rendering Characteristics" in the *XGL Programmer's Guide* for information on this environment variable.

The value returned by this function is used as the gamma value to build gamma and inverse gamma look-up tables. These tables are built by `buildGammaTables()`, which is called by the constructors for objects like `XglRasterWin` and `XglRasterMem`. The gamma and inverse gamma tables are only used for manipulating the colors of antialiased stroke and dot primitives. If a device implements antialiasing in hardware, then these tables, and hence the `getGammaValue()` function have no effect. However, if the device expects stroke or dot antialiasing to be done by the software pipeline, there are two possible cases. First, if the device does its own gamma correction, the function needs to return the value 1.0. Otherwise, the device can choose to not implement this function or to implement it to return a gamma value that is more suitable.

Note that this function might not be present in future releases of XGL; check the current header files for the most up-to-date list of optional functions.

### *Overridable Functions in DpDevRaster.h*

```
virtual void setRectList(const Xgl_irect[])
```

Sets the list of clip rectangles in the application-specified clip list. The input argument is an *Xgl\_irect* array of rectangles that defines the clip region. This function maps to the API attribute `XGL_RAS_RECT_LIST` and is used by the XGL core to inform the pipeline when the clip list changes. The default is no operation. See the `XGL_RAS_RECT_LIST` reference page for more information.

```
virtual void setRectNum(Xgl_usgn32)
```

Sets the number of clip rectangles in the application-specified clip list. The input argument is an unsigned 32-bit integer. This function maps to the API attribute `XGL_RAS_RECT_NUM` and is used by the XGL core to inform the pipeline when the clip list changes. The default is no operation. See the `XGL_RAS_RECT_NUM` reference page for more information.

```
virtual void setSourceBuffer(Xgl_buffer_sel)
```

Specifies the buffer used as the source buffer during raster operations. The input argument is a macro value from the *Xgl\_buffer\_sel* typedef. This function maps to the API attribute `XGL_RAS_SOURCE_BUFFER` and is used by the XGL core to inform the pipeline when the source buffer for raster operations changes. The default is no operation. See the `XGL_RAS_SOURCE_BUFFER` reference page for more information.

```
virtual void setSwZBuffer(XglPixRectMem*)
```

Specifies that if the device uses a software Z-buffer, it should share it with the base device in the backing store device by getting the memory address and linebytes from the Z-buffer and reassigning them as its own. This function is called by the XGL core when the device is a backing store device, so the device pipelines do not need to check for it. The default is no operation.

```
virtual void setSwAccumBuffer(XglPixRectMem*)
```

Specifies that if the device uses a software accumulation buffer, it should share the same software accumulation buffer with the base device in the backing store device by getting the memory address and linebytes from the

accumulation buffer and reassigning them as its own. The XGL core calls this function when the device is a backing store device, so the device pipelines do not need to check for it. The default is no operation.

```
virtual void syncRtnDevice(XglRasterWin*)
```

Synchronizes any device-dependent attributes, if needed, for backing store devices. The default is no operation.

## *Overridable Functions in DpDevWinRas.h*

```
virtual Xgl_usgn32 getDepth()
```

Returns the number of bits required to store the color of one pixel in the image buffer of this hardware device. The default behavior is to query the Drawable object for the depth of the frame buffer image buffer.

```
virtual Xgl_accum_depth getAccumBufferDepth()
```

Returns the accumulation buffer depth supported by the device. Called by the XGL core during the creation of the XglDpDev object. This function is currently only used by the software pipeline when doing accumulation.

```
virtual Xgl_color_type getRealColorType()
```

Returns the real color type of the device. The default behavior is to query the Drawable object for the real color type of the frame buffer.

```
virtual void resize()
```

Called by the XGL core when the pipeline's window is resized. The default is no operation.

```
virtual void setBackingStore(Xgl_boolean)
```

Requests backing store support from the device. No device pipeline operation is needed if the device relies on the XGL core to handle backing store manipulation. The input argument is a Boolean that indicates the on/off setting for backing store. This function maps to the API attribute XGL\_WIN\_RAS\_BACKING\_STORE; see the XGL\_WIN\_RAS\_BACKING\_STORE reference page for more information.

```
virtual Xgl_usgn32 setBuffersRequested(Xgl_usgn32)
```

Defines the number of buffers requested by the application. This function maps to the API attribute XGL\_WIN\_RAS\_BUFFERS\_REQUESTED and is used by the XGL core to request single or double buffering for the device. The default return value is one buffer.



```
virtual void setBufDraw(Xgl_usgn32)
```

Specifies the current draw buffer. This function maps to the API attribute `XGL_WIN_RAS_BUF_DRAW` and is used by the XGL core to set the current draw buffer. The default is no operation. See the `XGL_WIN_RAS_BUF_DRAW` reference page for more information.

```
virtual void setBufDisplay(Xgl_usgn32)
```

Specifies the current display buffer. This function maps to the API attribute `XGL_WIN_RAS_BUF_DISPLAY` and is used by the XGL core to set the current display buffer for the device. The default is no operation. See the `XGL_WIN_RAS_BUF_DISPLAY` for more information.

```
virtual void setBufMinDelay(Xgl_usgn32)
```

Defines the minimum time delay between buffer switches for this device. This function maps to the API attribute `XGL_WIN_RAS_BUF_MIN_DELAY`. The default is no operation. See the `XGL_WIN_RAS_BUF_MIN_DELAY` reference page for more information.

```
virtual void setCmap(XglCmap*)
```

Sets the color map. This function maps to the API attribute `XGL_DEV_COLOR_MAP` and is used by the XGL core to inform the pipeline when the XGL Color Map object changes. The input argument is a pointer to a Color Map object. The default is no operation.

When the contents of the Color Map change, `XglDpDevWinRas::setCmap()` is called. The device pipeline `messageReceive()` is called for object type `XGL_WIN_RAS` with message flag `XGLI_MSG_DEV_COLOR`. This message tells the device pipeline to handle the appropriate plane mask and color map changes.

```
virtual void setPixelMapping(const Xgl_usgn32[])
```

Sets the pixel mapping from the application's color indexes to the device color indexes. This function maps to the API attribute `XGL_WIN_RAS_PIXEL_MAPPING` and is used by the XGL core to inform the device when the pixel mapping changes. The input argument is an array of color values. The default is no operation. See the `XGL_WIN_RAS_PIXEL_MAPPING` reference page for more information.

```
virtual void setStereoMode(Xgl_stereo_mode)
```

Requests stereo mode support from the device. The input argument is an `Xgl_stereo_mode` enumerated value for the stereo setting. This function maps to the API attribute `XGL_WIN_RAS_STEREO_MODE` and is used by the XGL core to set the stereo mode on the device. The default is no operation. See the `XGL_WIN_RAS_STEREO_MODE` reference page for more information.

```
virtual XglPixRectMem* getSwZBuffer()
```

Returns a pointer to the `XglPixRectMem` object that represents the software Z-buffer. This function should be overridden by all devices that have a software implementation of the Z-buffer. The default return is `NULL`, meaning that if the device has a hardware Z-buffer, it does not need to override this function.

```
virtual XglPixRectMem* getSwAccumBuffer()
```

Returns a pointer to the `XglPixRectMem` object that represents the software accumulation buffer. This function should be overridden by all devices that have a software implementation of the accumulation buffer. The default return is `NULL`, meaning that if the device has a hardware accumulation buffer, it does not need to override this function.

```
virtual Xgl_boolean needRtnDevice()
```

Returns `TRUE` if the base device needs a shadow device for backing store. Device pipelines that provide for backing-store support in hardware will override this function, as will Xlib or PEXlib pipelines that use backing-store support in the server.

## *Overridable Functions in DpDevMemRas.h*

---

**Note** – For all practical purposes, there is only one Memory Raster pipeline, which is provided by XGL. The device pipeline does not need to override the functions in `DpDevMemRas.h`, as they are overridden in XGL's Memory Raster pipeline.

---

```
virtual XglPixRectMem* getImageBufferPixRect()
```

Returns a pointer to the `XglPixRectMem` object that represents the image buffer for the memory raster. The default return is `NULL`.

```
virtual XglPixRectMem* getZBufferPixRect()
```

Returns a pointer to the XglPixRectMem object that represents the Z-buffer for the memory raster. The default return is NULL.

```
virtual XglPixRectMem* getAccumBufferPixRect()
```

Returns a pointer to the XglPixRectMem object that represents the accumulation buffer for the Memory Raster. The default return is NULL.

```
virtual Xgl_accum_depth getAccumBufferDepth()
```

Returns a value for the depth of the accumulation buffer. The default return value is XGL\_ACCUM\_DEPTH\_2X.

```
virtual void setCmap(XglCmap*)
```

Sets the color map. This function maps to the API attribute XGL\_DEV\_COLOR\_MAP and is used by the XGL core to inform the pipeline when the XGL Color Map object changes. The input argument is a pointer to the Color Map object.

```
virtual void setImageBufferAddr(Xgl_usgn32*)
```

Specifies the array of pixels used in an XGL Memory Raster. This function maps to the API attribute XGL\_MEM\_RAS\_IMAGE\_BUFFER\_ADDR. See the XGL\_MEM\_RAS\_IMAGE\_BUFFER\_ADDR reference page for more information.

```
virtual void setZBufferAddr(Xgl_usgn32*)
```

Sets the starting address of the block of memory for the Z-buffer of a Memory Raster. This function maps to the API attribute XGL\_MEM\_RAS\_Z\_BUFFER\_ADDR. See the XGL\_MEM\_RAS\_Z\_BUFFER\_ADDR reference page.

```
virtual void setLineBytes(Xgl_usgn32)
```

Sets the linebytes value when the memory raster is set up to access memory for retained windows. linebytes is the number of bytes that separates one line in a raster, in other words, the number of bytes from (x,y) to (x,y+1).

## *What Else You Should Know*

This section provides additional information about the pipeline framework classes that you might need to know as you set up your device pipeline.

### *How a Device Pipeline Is Loaded*

A device pipeline is loaded when the application calls `xgl_inquire()` or calls `xgl_object_create()` to create a Device object for the first time. With the device object creation call, the application passes in the device type and the descriptor containing the X window identifying information. The XGL core then proceeds to create the objects needed for the pipeline. The pipeline loading part of this process is as follows:

1. When the application requests a Device object with `xgl_object_create()`, the System State object initiates the creation of the device-independent part of the Device object. In the case of a window raster, `XglRasterWin` is created.
2. When `XglRasterWin` is created, it calls `XglDrawable::grabDrawable()` to obtain an `XglDrawable` object of the appropriate type for the XGL device.
3. During the creation process of `XglRasterWin`, the Device object asks the `XglGlobalState` object to create its device-dependent part.
4. In the process of creating the device-dependent part of the Device object, the `XglGlobalState` object does the following:
  - a. It gets the name of the pipeline shared object file by calling the Window Raster's `XglDrawable::getPipeName()` routine. This routine issues a system call to the kernel device driver, using the `VIS_GETIDENTIFIER ioctl()`, to get a string specifying the name of the device when the device is a frame buffer. This string can then be used to create the pipeline name, which is of the form:
 

```
xgl<COMPANY NAME><device name>.so.<major version>
```
  - b. The `XglGlobalState::getDpLib()` routine then traverses the Global State's `XglDpLibList` object list to determine if an object for the pipeline library already exists. If so, it returns.

If `XglGlobalState` does not find a match in its `XglDpLibList`, `XglGlobalState::loadPipeLib()` loads the pipeline using `dlopen()`, which is an interface routine in the Solaris dynamic linking mechanism. `dlopen()` gives XGL runtime access to the device pipeline shared object file and binds it to XGL's process address space.

- c. `XglGlobalState::loadPipeLib()` then creates the `XglDpLib` object for the pipeline by calling the device pipeline's `xgl_create_PipeLib()` routine, which is defined in the pipeline and accessed through `dlsym()`. `xgl_create_PipeLib()` first checks the DDK major and minor version numbers to ensure that the pipeline is compatible with the core XGL library that is attempting to load it. If this check succeeds, `xgl_create_PipeLib()` creates an instance of the pipeline-derived `XglDpLib` class and returns a pointer to the object. This pointer is appended to the `XglGlobalState`'s `XglDpLibList` object for future reference. The `XglDpLib` object represents a single pipeline.

At this point, the process of pipeline object creation continues with the instantiation of the pipeline `XglDpMgr` object and the pipeline `XglDpDev` object. For detailed information on the complete set of steps that occur during pipeline creation, see the *XGL Architecture Guide*.

## Device Pipeline Objects for Multiple Processes

The device pipeline objects in Figure 3-2 are created when a pipeline is instantiated on a single frame buffer from a single application.



Figure 3-2 Pipeline Objects for a Single Application

When a single application opens windows on a two-headed system in which the frame buffers are the same type, there is one `XglDpLib` object, an `XglDpMgr` for each frame buffer, and an `XglDpDev` object for each window. The single application program is one UNIX process. A diagram of a single application opening one window on each of two identical frame buffers would look like Figure 3-3 on page 60.

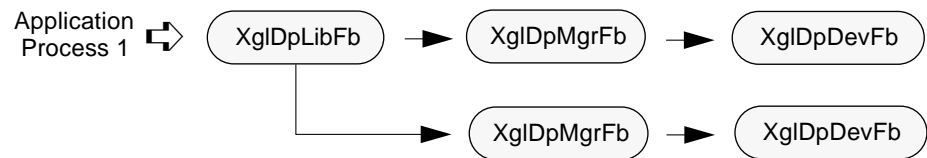


Figure 3-3 Pipeline Objects for an Application on Multiple Frame Buffers

When there is more than one application program using XGL, there is a UNIX process for each application. If there are two application programs, there are two UNIX processes. In this case, there is an XglDpLib object for each process, an XglDpMgr object corresponding to each XglDpLib, and an XglDpDev object for each window. Figure 3-4 shows the pipeline objects that are created for two application programs running on one frame buffer. The second application opens two windows.

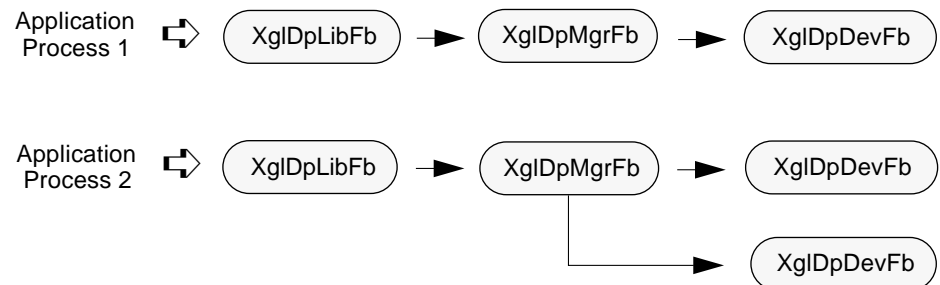


Figure 3-4 Pipeline Objects for Two Applications

The XGL/DGA system does not describe how an accelerator accommodates two or more application processes. DGA is basically a concurrency control mechanism; it will serialize concurrent accesses, but it does not mandate how the accelerator handles state information for different processes. You must coordinate the interaction between the XglDpLib objects for each process.

If the application programs run on a two-headed system with frame buffers of different types, for example, your frame buffer and a GX frame buffer, the pipeline objects that are created might look like Figure 3-5 on page 61.

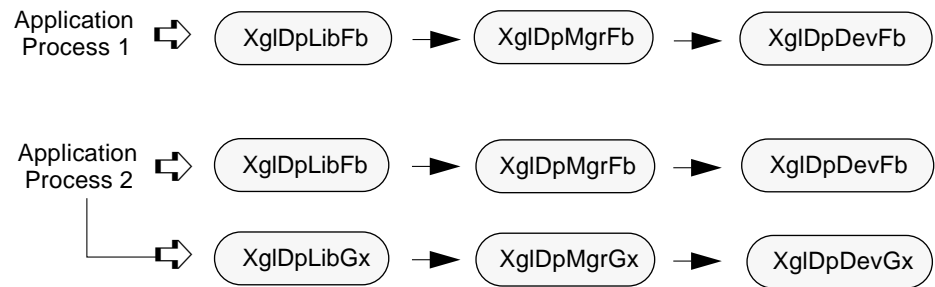


Figure 3-5 Pipeline Objects for Applications on Multiple Frame Buffers

## Adding Member Data to a Pipeline Class

When creating subclassed pipeline classes, you can add member data whenever needed. The following example illustrates a way to add and initialize a pointer to the device manager as member data in the `XglDpDev` and `XglDpCtx` subclasses.

1. First, add a device pipeline manager pointer as member data and a device pipeline manager parameter to the constructors of `XglDpDevSampDp` and `XglDpCtx[2/3]dSampDp`:

```

class XglDpDevSampDp : public XglDpDevWinRas {
    friend XglDpLibSampDp;
private:
    // call base class constructor with device,
    // assign device manager pointer (dev_mgr)
    // to member data dpMgr
    XglDpDevSampDp(XglDevice* device, XglDpMgrSampDp* dev_mgr)
        : XglDpDevWinRas(device), dpMgr(dev_mgr) { }

    XglDpMgrSampDp* dpMgr;    // device manager pointer
    //      .... other declarations
};

class XglDpCtx3dSampDp : public XglDpCtx3d {
    friend XglDpDevSampDp;
private:
    // call base class constructor with context,
    // assign device manager pointer (dev_mgr)

```

```
class XglDpDevSampDp : public XglDpDevWinRas {
    // to member data dpMgr
    XglDpCtx3dSampDp(XglContext3d* context, XglDpMgrSampDp*
                    dev_mgr)
    : XglDpCtx3d(context), dpMgr(dev_mgr)
    {
        opsVec[XGLI_LI1_MULTIPOLYLINE] =
            XGLI_OPS(XglDpCtx3dSampDp::li1MultiPolyline);

        // Attribute changes
        opsVec[XGLI_LI_OBJ_SET] =
            XGLI_OPS(XglDpCtx3dSampDp::objectSet);

        // Message handler
        opsVec[XGLI_LI_MSG_RCV] =
            XGLI_OPS(XglDpCtx3dSampDp::messageReceive);
    }
    XglDpMgrSampDp* dpMgr;           // device manager pointer
    // .... other declarations
};
```

## 2. Modify the object-creation functions to pass the device manager pointer to the constructors:

```
XglDpDev* XglDpMgrSampDp::createDpDev(XglDevice* device)
{
    // "this" is the device manager (XglDpMgrSampDp) itself
    return new XglDpDevSampDp(device, this);
}

XglDpCtx3d* XglDpDevSampDp::createDpCtx(XglContext3d* context)
{
    // here dpMgr is a member data of XglDpDevSampDp
    return new XglDpCtx3dSampDp(context, dpMgr);
}
```



## Versioning

The XGL core library (`libxgl.so`) dynamically loads device pipeline modules at runtime; therefore, a versioning scheme is required to ensure that the core library and the pipeline that it loads are compatible. The versioning scheme is implemented both as part of the XGL core library and as part of the Driver Developer Kit (DDK).

The DDK contains header files that define the interfaces between the core XGL library and the dynamically loaded pipeline modules. The core library and the DDK have a version number that is called the DDK version number. This version number, which contains both major and minor parts, is defined by two macro definitions in the file `xgli/DdkVersion.h`. The macro definitions for the current release are:

```
#define XGLI_DDK_MAJOR_VERSION 4
#define XGLI_DDK_MINOR_VERSION 0
```

Every XGL device pipeline must include the `DdkVersion.h` header file in order to use the versioning information.

## Versioning Rules

Each release of XGL is accompanied by a corresponding release of the DDK containing files used to build the core XGL library and the reference device pipelines. Independent Hardware Vendors (IHV's) use the DDK to build a device handler that is compatible with the core XGL library in that release.

The DDK version number is unrelated to the XGL API library version number. For example, the 3 in `libxgl.so.3` is the version number of the XGL API release. It is not related to the internal DDK `majorVersion` number. IHV's supplying XGL device pipelines must conform to the following versioning rules:

1. The DDK `majorVersion` (defined in `xgli/DdkVersion.h`) used to build the device pipeline is stamped in the file name of the device pipeline, such as, `xglSUNWcg6.so.4`, where the 4 is the same as `majorVersion`. The convention used to name the device pipelines is:

```
xgl<COMPANY NAME><device name>.so.<majorVersion>
```

2. The core XGL library is stamped internally with both the DDK major and minor version numbers of the DDK used to build it. The core XGL library will never load a pipeline with a DDK `majorVersion` greater than its own. For example, `libxgl.so` with DDK internal version number 3 will not load a pipeline named `xglSUNWcg6.so.4`.
3. The core XGL library will load a device pipeline with a DDK `majorVersion` less than its own DDK `majorVersion` only if the XGL core library has explicitly decided to emulate that lesser `majorVersion` interface. Every time a new version of XGL and the XGL DDK are released, this DDK document will specify which, if any, DDK major versions are emulated by the core XGL library.

For this release of the DDK (major version 4, minor version 0), no prior versions are emulated.

4. The core XGL library will always attempt to dynamically load a device pipeline that has the same DDK `majorVersion` as itself. If the device pipeline depends on functionality that was added in a particular `minorVersion` of the DDK, your pipeline must check for the existence of that functionality by checking the core library's DDK version number.

A device pipeline can provide its own workaround if the functionality does not exist, or it can fail with an appropriate error message indicating the core library version that is required.

The functionality differences between `minorVersion` releases of the DDK will be documented in the DDK documentation. A device pipeline can check the core library's DDK version number by calling the global library function `xglGetDdkVersion()` from within its `xgl_create_PipeLib()` function, as declared in `xgli/DdkVersion.h`.

There is also an `XGLI_PIPELINE_CHECK_VERSION()` macro in the `DdkVersion.h` file which shows a sample implementation. A device pipeline is free to use it or implement its handling in the `xgl_create_PipeLib()` routine as long as it adheres to the versioning rules stated above.

```
#include "xgli/DdkVersion.h"

//
// Interface routine called after pipeline loading
//

XglPipeLib* xgl_create_PipeLib()
{
    XGLI_PIPELINE_CHECK_VERSION(XglDpLibSampDp);
}
```

## ***Backing Store Support in the Pipeline Classes***

The XGL core is responsible for handling XGL backing store device creation and use. The device pipeline needs only to implement a small set of device-dependent functions in certain cases. This section summarizes the functions that the device pipeline needs to implement. For more information on how the XGL core handles backing store, see the *XGL Architecture Guide*.

### ***Backing Store Clipping Status Values***

If the device pipeline can determine whether a primitive is clipped, it can notify the device-independent layer with the `setPrimClipStatus()` function to indicate the current status. The following argument values are defined in `DpCtx.h`:

<code>XGLI_DP_STATUS_FAIL</code>	The primitive could not be rendered.
<code>XGLI_DP_STATUS_SUCCESS</code>	The primitive was successfully rendered and may or may not have been clipped.
<code>XGLI_DP_STATUS_FULLY_RENDERED</code>	The primitive was successfully rendered without being clipped.

You can use the value `XGLI_DP_STATUS_FULLY_RENDERED` for all the primitives at the LI-1 level.

The `XGLI_DP_STATUS_FULLY_RENDERED` value means that the primitive was successfully rendered, *and* that the primitive was fully rendered into the window without any clipping. This argument value is optional and applies only to synchronous accelerators (those without queues). If the graphics device cannot determine whether the primitive is clipped, it is not necessary to call `setPrimClipStatus()`.

The `XGLI_DP_STATUS_FULLY_RENDERED` value is an optimization to improve the performance of applications using backing store when the window is partially covered. If a device pipeline can set this status, performance is increased if there is a backing-store device and if the window is partially covered. This optimization does not apply to accelerators that cannot determine the clip status.

---

**Note** – The device pipeline should never set the value `XGLI_DP_STATUS_UNCLIPPED` (defined in `DpCtx.h`). This value is for internal use only.

---

## *Backing Store in Window Raster Pipelines*

The following functions in `XglDpDevWinRas.h` should be overridden by all devices that provide Z-buffers or accumulation buffers in software. See page 54 for information on these functions.

- `virtual XglPixRectMem* getSwZBuffer()`
- `virtual XglPixRectMem* getSwAccumBuffer()`

Device pipelines that can handle backing store in hardware or the X11 server (for example, the PEXlib pipeline) will override the following function. A pipeline that returns `FALSE` can ignore the remainder of the functions in this section.

- `virtual Xgl_boolean needRtnDevice()`

### *Backing Store Support for Backing Store Devices*

Devices that provide backing store support, such as Memory Raster devices or a hardware device with a cache for backing-store memory, will override these functions in `XglDpDevRaster.h`. See page 53 for information.

- `virtual void setSwZBuffer(XglPixRectMem*)`
- `virtual void setSwAccumBuffer(XglPixRectMem*)`
- `virtual void syncRtnDevice(XglRasterWin*)`

### *Backing Store Support in the Dp Manager*

The device pipeline manager object provides an object descriptor for the backing store device:

```
XglDpDev* XglDpMgrFb::createDpDev(XglDevice*,  
                                   Xgl_obj_desc* bkstore_desc=NULL);
```

When the XGL core creates a backing store device, the descriptor is passed in as follows:

```
bkstore_desc.win_ras.type = XGL_WIN_RAS_BACKING_STORE;  
bkstore_desc.win_ras.desc = (pointer to the parent device);
```

A device pipeline can ignore this parameter if appropriate.

---

**Note** – The XGL API cannot support backing store and double buffering at the same time. Even if your device can support both, there are issues regarding the synchronization of double buffering and backing store with the X11 server that are not resolved in the current release of the server. Therefore, an application backing store request is denied by the XGL core when double buffering is enabled. Thus, even if your pipeline supports both double buffering and backing store, the pipeline will not be called for backing store when double buffering is enabled. See the `XGL_WIN_RAS_BACKING_STORE` reference page for details.

---

## Quick Reference Chart of Overridable Functions

In the device pipeline classes there are some functions that your device pipeline must override and other functions that are optional. Whenever possible, XGL provided defaults for functions; however, you will probably want to override XGL's version of these functions if your device can accelerate the functionality. Required functions are completely device dependent.

Table 3-2 provides a quick reference summary of all the pipeline functions; those marked "Required" must be overridden by the device pipeline, or an error will be returned. For information on input arguments and return values for the functions in the device pipeline classes, see page 52. For information on the LI-1, LI-2, and LI-3 loadable interfaces, see Chapter 9, "Writing Loadable Interfaces"..

Table 3-2 Summary of Optional and Required Pipeline Functions

Class	Function Name	Status
Device pipeline .so file	<code>xgl_create_PipeLib()</code>	Required
<b>XglDpLib</b>	<code>getDpMgr()</code>	Required
<b>XglDpMgr</b>	<code>createDpDev()</code>	Required
	<code>inquire()</code>	Required
<b>XglDpDev</b>	<code>createDpCtx()</code> for 2D	Required
	<code>createDpCtx()</code> for 3D	Required
	<code>copyBuffer()</code> for 2D	Required
	<code>copyBuffer()</code> for 3D	Required
	<code>getDcOrientation()</code>	Optional
	<code>getMaxZ()</code>	Optional
	<code>getGammaValue()</code>	Optional
<b>XglDpDevRaster</b>	<code>setRectList()</code>	All
	<code>setRectNum()</code>	optional
	<code>setSourceBuffer()</code>	
	<code>setSwZBuffer()</code>	
	<code>setSwAccumBuffer()</code>	
	<code>syncRtnDevice()</code>	

Table 3-2 Summary of Optional and Required Pipeline Functions (Continued)

Class	Function Name	Status
<b>XglDpDevWinRas</b>	getDepth()	All optional
	getAccumBufferDepth()	
	getRealColorType()	
	resize()	
	setBackingStore()	
	setBufDisplay()	
	setBufDraw()	
	setBufMinDelay()	
	setCmap()	
	setPixelMapping()	
	setStereoMode()	
	setBuffersRequested()	
	getSwZBuffer()	
	getSwAccumBuffer()	
	needRtnDevice()	
<b>XglDpDevMemRas</b>	getImageBufferPixRect()	All optional
	getZBufferPixRect()	
	getAccumBufferPixRect()	
	getAccumBufferDepth()	
	setCmap()	
	setImageBufferAddr()	
	setZBufferAddr()	
<b>XglDpCtx2d</b> LI-1 Primitives	setLineBytes()	All optional
	lilAnnotationText()	
	lilDisplayGcache()	
	lilMultiArc()	
	lilMultiCircle()	
	lilMultiMarker()	
	lilMultiPolyline()	
	lilMultiRectangle()	
	lilMultiSimplePolygon()	
	lilNurbsCurve()	
	lilPolygon()	
	lilStrokeText()	

**Table 3-2** Summary of Optional and Required Pipeline Functions (Continued)

<b>Class</b>	<b>Function Name</b>	<b>Status</b>
Pixel and Raster Operators	li1NewFrame	Required
	li1CopyBuffer()	Required
	li1GetPixel()	Required
	li1Image()	Optional
	li1SetMultiPixel()	Optional
	li1SetPixel()	Required
	li1SetPixelRow()	Optional
	li1Flush()	Optional
	li1PickBufferFlush()	Optional
LI-2 Functions	li2GeneralPolygon	All optional
	li2MultiDot()	
	li2MultiEllipse()	
	li2MultiEllipticalArc()	
	li2MultiPolyline()	
	li2MultiRect()	
	li2MultiSimplePolygon()	
LI-3 Functions	li3Begin()	All required
	li3End()	
	li3MultiDot()	
	li3Vector()	
	li3MultiSpan()	
	li3CopyFromDpBuffer()	
	li3CopyToDpBuffer()	
State Changes	objectSet()	Required
Message Passing	messageReceive()	Required
<b>XglDpCtx3d</b>		
LI-1 Primitives	All 2D primitives and the following:	All optional
	li1MultiEllipticalArc()	
	li1NurbsSurf()	
	li1QuadrilateralMesh()	
	li1TriangleList()	
	li1TriangleStrip()	
Pixel and Raster Operators	All 2D pixel and raster functions and the following:	Optional
	li1Accumulate()	
	li1ClearAccumulation()	



Table 3-2 Summary of Optional and Required Pipeline Functions (Continued)

Class	Function Name	Status
LI-2 Functions	li2GeneralPolygon()	All optional
	li2MultiDot()	
	li2MultiPolyline()	
	li2MultiSimplePolygon()	
	li2TriangleList()	
	li2TriangleStrip()	
LI-3 Functions	li3Begin()	All required
	li3End()	
	li3MultiDot()	
	li3Vector()	
	li3MultiSpan()	
	li3CopyFromDpBuffer()	
	li3CopyToDpBuffer()	
State Changes	objectSet()	Required
Message Passing	messageReceive()	Required



## *Internal Data Storage*

---

4 

This chapter describes the internal data types that XGL uses to transfer information from one part of the pipeline to another. It explains how the data types are constructed and shows some examples of their use. The chapter includes information on the following topics:

- Data storage in the `XglPrimData` object
- Accessing partially processed geometry data in `XglPrimData` at the LI-2 level
- Conic and rectangle data storage in the `XglConicData2d`, `XglConicData3D`, `XglRectData2d`, and `XglRectData3d` objects
- Accessing pixel data in the `PixRect` object.



As you read this chapter, you will find it helpful to have access to the following header files:

- `PrimData.h`
- `RectData2d.h` and `RectData3d.h`
- `ConicData2d.h` and `ConicData3d.h`
- `PixRect.h`, `PixRect2d.h`, and `PixRect3d.h`

## *Internal Data Types*

The XGL API offers a wide variety of point types that applications can use to pass data to XGL. Although varied, these point types do not contain all the information that a device pipeline might need to efficiently display the data. To solve this problem, XGL designed a number of internal data types that the pipeline can reference to get application data. These internal data types contain both the application geometry and some useful information about the geometry.

At the LI-1 layer, the API parameters are passed directly to the device pipeline. For more information on the LI-1 API arguments, refer to the *XGL Reference Manual*.

Most geometry in XGL that is passed to and between loadable interface layer 2 is stored under the control of a C++ class called `XglPrimData`. This class contains pointers to the original API data (essentially the arguments to the API primitive), together with a framework that is used internally by the software pipeline. `XglPrimData` is used to handle both 2D and 3D point and facet types; the same structures are used in both cases.

Although `XglPrimData` is the input to many of the rendering functions in LI-2, it is not used for rendering conics (circles, arcs, ellipses, or elliptical arcs) or rectangles. Conic data is stored in either the `XglConicData2d` object or the `XglConicData3d` object. Similarly, rectangle data is stored in the `XglRectData2d` object or the `XglRectData3d` object. These objects are similar to `XglPrimData`. The sections that follow discuss accessing data from these objects and from `XglPrimData`.

At the LI-3 layer, a different set of data structures is used for point data. These data structures, along with the LI-3 functions that use them, are described in Chapter 9, “Writing Loadable Interfaces”. To help you implement the LI-3 layer, XGL provides a set of utilities that render to a device via the `PixRect` object. A `PixRect` is an abstraction of a rectangular array of pixels that represents the underlying frame buffer. Information on the `PixRect` object and its interfaces is provided in this chapter on page 89.

## Accessing Data at the LI-1 Layer

To access data at the LI-1 layer, the device pipeline receives the API data directly from the application. Figure 4-1 illustrates the flow of data through the LI-1 primitive.

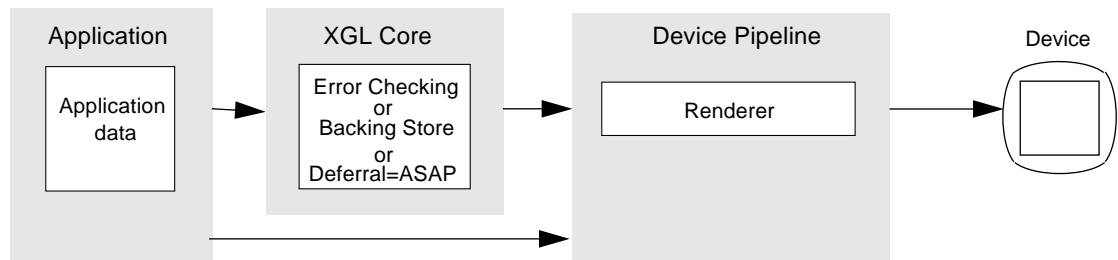


Figure 4-1 Flow of Application Data Through the LI-1 Primitive

This section presents several code samples for accessing application data. You can copy or modify these source code samples and the other source code samples in this chapter as long as the resulting code is used to create a loadable pipeline for XGL.

## Accessing Application Data

The example below shows how a fictitious device might access data from the application to draw a simple 2D multipolyline.

```

void XglDpCtx2dExample::lilMultiPolyline(
    Xgl_bbox*      bounding_box,
    Xgl_usgn32     num_pt_lists,
    Xgl_pt_list    *pl);
{
    //
    // NOTE: This example assumes a point type of Xgl_pt_f2d
    //
    Xgl_pt_f2d*pt;
    int      num_pts;

    //
    // Loop through the point lists.
    //
  
```

```

for ( ; num_pt_lists>0 ; num_pt_lists--, pl++) {

    //
    // Check for at least 2 vertices per point list
    //
    if ((num_pts = pl->num_pts) < 2)
        continue;

    //
    // Send all vertices to hardware
    //
    for ( pt=pl->pts.f2d ; num_pts>0 ; num_pts--, pt++ ) {

        send_xcoord_to_hardware(pt->x);
        send_ycoord_to_hardware(pt->y);

    }

}

```

## Accessing Facet Data

This example shows how to access facet data for 3D surfaces, as well as how to access other data. This example assumes that `xgl_multi_simple_polygon()` has been called with facet colors and that lighting is enabled. This requires that the fictitious device send down a color for each facet, as well as a normal for each vertex.

```

void XglDpCtx3dExample::lilMultiSimplePolygon(
    Xgl_facet_flags    flags,
    Xgl_facet_list     *facets,
    Xgl_bbox           *bbox,
    Xg_usgn32          num_pt_lists,
    Xgl_pt_list        pl);
{
    //
    // NOTE: This example assumes a point type of
    // Xgl_pt_normal_f3d and a facet type of Xgl_color_facet
    //

    int    num_pts;
    Xgl_color *fc=facets->facet.color_facets;

    //
    // Loop through all the point lists

```

```

//
for ( ; num_pt_lists>0 ; num_pt_lists--, pl++ , fc++ ) {

    //
    // Check for at least 3 vertices per point list
    //
    if ((num_pts = pl->num_pts) < 3)
        continue;

    //
    // Set the color for the next facet.
    //
    send_rcolor_to_hardware(fc->color.rgb.r);
    send_gcolor_to_hardware(fc->color.rgb.g);
    send_bcolor_to_hardware(fc->color.rgb.b);

    //
    // Send all vertices and normals to hardware
    //
    for ( pt=pl->pts.normal_f3d ; num_pts>0 ; num_pts--,
          pt++ ) {

        send_xnorm_to_hardware(pt->normal.x);
        send_ynorm_to_hardware(pt->normal.y);
        send_znorm_to_hardware(pt->normal.z);

        send_xcoord_to_hardware(pt->x);
        send_ycoord_to_hardware(pt->y);
        send_zcoord_to_hardware(pt->z);

    }
}

```

### *Point Lists with Data Mapping Values*

When data-mapping values are present in the point list, the point size is equal to the sum of the sizes of all of the fields as mentioned above, but only one of the data values is accounted for. For instance, if the point type is *Xgl\_pt\_data\_f3d*, and there are three data values per point, then the point size is 16 (x,y,z = 12 bytes, plus 4 bytes for the first data value). To calculate the correct point size the following equation must be used:

```

true_point_size = point_size + (num_data_values - 1)*sizeof(float)

```

The number of data values is recorded in a field of *Xgl\_pt\_list*. The reason this extra calculation is necessary is that some primitives (like multisimple polygon) take an array of point lists as an argument. The number of data values per vertex in each list can be different; thus, the point size can be different for each list.

## Data Access for DMA Devices

The next example shows how a device that uses direct memory access (DMA) might access data. Devices that use DMA to transfer data require only a starting point from which to copy the data and the size of the data block (together, perhaps, with some header block that describes the type of data).

The geometric information pointed to by the *Xgl\_pt\_list* structure is guaranteed to always be contiguous. This is true even if a device pipeline is being called back by the software pipeline. This means that this interface is appropriate for devices that use DMA to communicate data to their hardware or copy it across from the host as do the two previous examples.

```
void XglDpCtx2dDmaExample::lilMultiPolyline(
    Xgl_bbox*      bbox,
    Xgl_usgn32     num_pt_lists,
    Xgl_pt_list*   pl);
{
    // NOTE: This example assumes a point type of Xgl_pt_f2d
    //
    // Loop through the point lists
    //
    for ( ; num_pt_lists>0 ; num_pt_lists--, pl++) {

        //
        // Check for at least 2 vertices per point list
        //
        if (pl->num_pts < 2)
            continue;
        // Send all vertices to hardware
        //
        wait_for_outstanding_dma_to_finish();
        send_dma_ptlist_to_hardware(pl->pts.f2d,
            pl->num_pts*sizeof(Xgl_pt_f2d));
    }
}
```



## How Data Is Stored by the Software Pipeline

The `XglPrimData` class includes a subclass called `XglLevel`. This class is used by the software pipeline to store the results of processing the original API data. A *level* is a memory area for storing primitive data. Each time the data is modified, as it would be after transformations, clipping, lighting, depth cueing, shading, or texture mapping, a new level is started. This allows the software pipeline to move data around as it processes data and provides it with access to previous stages of the pipeline. It also allows a device pipeline to refer back to an earlier version of the data. Figure 4-2 illustrates the `XglLevel` objects that would be created for a hypothetical software pipeline that transformed, clipped, and lit the geometry data; level 0 contains the original API data.

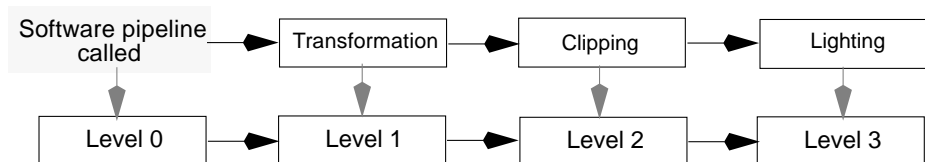


Figure 4-2 Level Objects Created by Software Pipeline Processing

The `XglPrimData` class maintains an array of `XglLevel` objects. This is effectively a stack, with each object representing the data in various stages of processing. The bottommost `XglLevel` object, level 0, contains the API data, while the topmost object contains the processed geometry. In an LI-2 renderer the data to be used is read out from this topmost object. Figure 4-3 on page 80 illustrates the flow of data from the application to a device pipeline that is ported at the LI-2 layer.

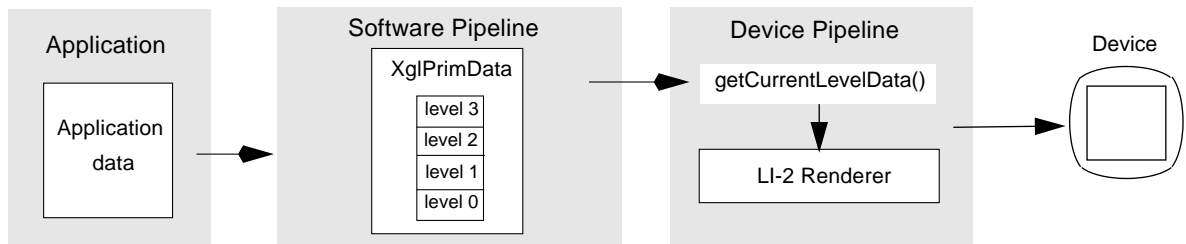


Figure 4-3 Flow of Point Data Through XglPrimData and XglLevel

### Data Storage in the XglLevel Object

The XglLevel class stores data in a noncontiguous format. This is done by specifying a base-pointer and step-size pair for each field in the point that is being processed. The base pointer points to the field for the first point in the list. The step size indicates how many bytes to increment the pointer to get to the field in the second point (and so on).

Initially, the base pointers all point to the beginning of the API data, and the step sizes are all the same, in other words, equal to the point size. Graphically, this would look something like Figure 4-4, assuming a point type that contained geometry, colors, and normals.

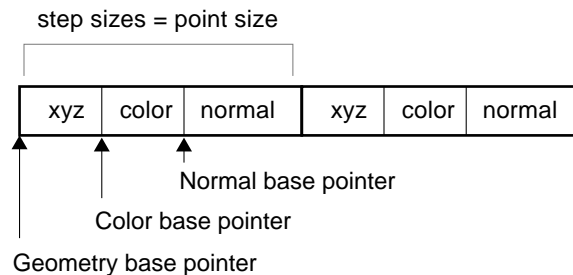


Figure 4-4 Base/Offset Data Storage in XglLevel

Thus, to get to the color field of the second point, the color base pointer would be incremented by the point size.

During normal operation of a software LI-1 routine, one or more of these pointers is replaced by a pointer to a different area of memory, local to XGL. The step sizes are adjusted accordingly. For instance, starting from the sets of pointers and step sizes pictured above, the geometry values may be transformed, and the results stored to a different area of memory. This would change the picture to something like Figure 4-5.

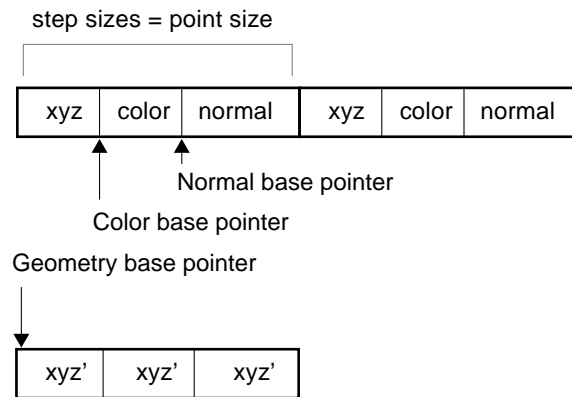


Figure 4-5 Base/Offset Data When the Point Data Has Changed

The geometry base pointer no longer points to the API data, but rather to an array of points local to the pipeline. Since the transformation did not affect the colors or the normals, their pointers still point to the API data. The new geometry step size will be equal to the size of [x,y,z] since the array contains no other information. This technique allows the software pipeline to process data efficiently, since only that data which is actually modified is copied. Unmodified data is left in its original form in the user's space.

In order to hold both the separate pointers and step sizes, an internal point list structure, *Xgli\_point\_list*, is used. This structure contains the data outlined above, in addition to some flags that control rendering, such as a close flag for polylines that joins the first and last vertices, and an indication of whether a 3D surface is front facing or back facing. See *XglPrimData.h* for the structures that make up *XglLevel*.

## *LI-2 Point Data*

LI-2 point data is stored internally under the control of the XglPrimData object. The information relevant to the device pipeline is stored in an object called XglLevel within the XglPrimData object. XglLevel contains point list information that is created when the data moves down through the software pipeline. For example, the first level, level 0, is created when the LI-1 software pipeline is called. If points are transformed, the transformed points are stored at a different level than the original points. Level objects are used extensively by the software pipeline and are the device pipeline LI-2 layer interface to the processed geometry.

### *Accessing Data at the LI-2 Layer*

Since the software pipeline makes use of the XglLevel structures in its LI-1 processing, any device pipeline LI-2 functions must extract data in XglLevel format from these structures. Level format means that all the point and facet lists have been broken down into base-pointer/step-size format, as shown in Figure 4-4 and Figure 4-5.

The methods for extracting data in level format use the XglPrimData method `getCurrentLevelData()`. This method provides offset and step-size information that is available from the structure directly which does not have to be computed.

The following code fragment is an example of how a device pipeline might implement an LI-2 polyline renderer.

```
XglDpCtx2dExample::li2MultiPolyline(XglPrimData *pd)
{
    //
    // First get the XglLevel structure. This method gets the
    // current level, that is the one that contains the most
    // up-to-date data.
    //
    level = pd->getCurrentLevelData();

    //
    // Get the number of point lists, and the point lists
    // themselves.
    //
    num_pl = level->getNumPointLists();
```

```

pl = level->getPointLists();

//
// See if we have to close the polylines. If this routine is
// being called to draw a hollow polygon, for instance, then
// the first and last points need to be connected.
//
close_flag = pd->getProcessFlags() & XGLI_CLOSE_FLAG;

//
// Loop on the point lists.
//
for (i = 0; i < num_pl; i++) {
    pt = (Xgl_pt_i2d*) pl->geom_ptr.base_ptr;

    //
    // Loop on the points in each point list.
    //
    for (j = 0; j < pl->current_num_points; j++) {
        send_to_hardware(pt->x);
        send_to_hardware(pt->y);
        XGLI_INCR(pt, Xgl_pt_i2d*, pl->geom_ptr.step_size);
    }

    //
    // Optionally close the polyline - send down the 1st pt
    // again.
    //
    if (close_flag) {
        pt = (Xgl_pt_i2d*) pl->geom_ptr.base_ptr;
        send_to_hardware(pt->x);
        send_to_hardware(pt->y);
    }
}
}

```

**Note** – The *lighting\_coeffA\_ptr*, *lighting\_coeffB\_ptr*, and *use\_lighting\_coeffs* fields in the *Xgli\_point\_list* and *Xgli\_facet\_list* structures used by *XglLevel* store the lighting coefficients on a per-vertex and per-facet basis when lighting is on and texturing is on. See Chapter 9, “Writing Loadable Interfaces” for more information on texture mapping.

## *Pipeline Interfaces to XglPrimData and XglLevel Data*

Table 4-1 lists XglPrimData interfaces that the device pipeline can use to get point data and to get information about point data.

*Table 4-1* XglPrimData Interfaces

Function	Description
<code>getLevelData()</code>	Returns the data for a specified level.
<code>getCurrentLevel()</code> <code>getCurrentLevelData()</code>	Return the data for the current level.
<code>getProcessFlags()</code>	Returns a value indicating which software pipeline processing steps (such as clipping or lighting) need to be done.

Table 4-2 lists useful interfaces from the XglLevel subclass of XglPrimData.

*Table 4-2* XglLevel

Function	Description
<code>getPointLists()</code>	Returns the API data point lists.
<code>getFacetList()</code>	Returns the API data facet lists.
<code>getNumPointLists()</code>	Returns the number of point lists.
<code>getRenderFlags()</code>	Returns API rendering flags.
<code>getFaceAttrs()</code>	Returns the front facing and back facing attributes.

## Conic and Rectangle Data

The conic and rectangle data at the LI-1 layer is defined at the API level. This section discusses the conic and rectangle data at the LI-2 layer.

The `XglConicData{2,3}d` and `XglRectData{2,3}d` data structures are used to hold information for rendering conics and rectangles. These data structures are based on `XglPrimData` in that they organize the data into levels and use a base-pointer/step-size technique. They differ from `XglPrimData` in that the level structures are used at LI-2.

The level data in `XglConicData` is contained in an array of objects of type `XglConicList{2,3}d` rather than an array of `XglLevel` objects. Each `XglConicList` object is a level for a stage of the software pipeline for the conic. The object contains pointers to a list of conic data for each of the items describing a circle, arc, or other conic geometry, as well as information on the number of conics. The API data is referenced at level 0.

Similarly, the level data in `XglRectData` is contained in an array of objects of type `XglRectList{2,3}d`. `XglRectList` has pointers to a list of rectangles specified in `Xgl_rect_list` as a base and offset. The base points to the first rectangle in the list and the offset specifies the step size to access the next rectangle. The `XglRectList` object also contains a value for the number of rectangles.

The following examples illustrate how the pipeline can retrieve information from these objects.

### Accessing Rectangle Data from `XglRectData`

This example shows how to access data from an `XglRectData2d` object.

```
void XglDpCtx2dExample::li2MultiRect(XglRectData2d* rd)
{
    XglRectList2d*      rlist;
    Xgl_usgn32          num_rects;    // number of rectangles
    Xgl_rect_i2d*       rectangle;

    //
    // Extract the list of rectangles from the data structure.
    //
    rlist = rd->getCurrentLevelData();
    num_rects = rlist->getNumRects();
}
```

```

rectangle = (Xgl_rect_i2d *) (rlist
                             ->cornerPoints.base_ptr);

//
// Loop through the list of rectangles.
//
for (long i = 0; i < num_rects; i++, rectangle++) {
    send_to_hardware(rectangle->corner_min.x);
    send_to_hardware(rectangle->corner_min.y);
    send_to_hardware(rectangle->corner_max.x);
    send_to_hardware(rectangle->corner_max.y);
}
}

```

## Accessing Conic Data from XglConicData

This example shows how to access data from an XglConicData2d object.

```

void XglDpCtx2dExample::li2MultiEllipse(XglConicData2d* cd)
{
    XglConicList2d*    conic_list;
    Xgl_usgn32         num_ells;           // number of ellipses
    Xgl_pt_flag_f2d*   center;
    Xgl_usgn32*        major_axis;
    Xgl_usgn32*        minor_axis;
    float*             rot_angle;
    Xgl_usgn32 center_step, major_axis_step,
                    minor_axis_step, rot_angle_step;
    Xgli_pointer*ptr;

    // Get conic data.
    conic_list = cd->getCurrentLevelData();
    num_ells = conic_list->getNumConics();

    // Get rotation angle and step increment size.
    ptr = conic_list->getRotAnglePtr();
    rot_angle = (float *)ptr->base_ptr;
    rot_angle_step = ptr->step_size;

    //
    // This device pipeline cannot handle rotated ellipses.
    // Punt to software pipeline if rotation angle is not 0 or
    // pi/2.
    //
    if ( ! (XGLI_EQUAL_ZERO(*rot_angle,
                          XGLI_ANGULAR_TOLERANCE)

```



```

        || XGLI_EQUAL_ZERO((*rot_angle) - M_PI_2,
        XGLI_ANGULAR_TOLERANCE)) ) {

        swp->li2MultiEllipse(cd);
        return;
    }

    // Get center and step increment size.
    ptr = conic_list->getCenterPtr();
    center = (Xgl_pt_flag_f2d *)ptr->base_ptr;
    center_step = ptr->step_size;

    // Get major axis and step increment size.
    ptr = conic_list->getMajorAxisPtr();
    major_axis = (Xgl_usgn32 *)ptr->base_ptr;
    major_axis_step = ptr->step_size;

    // Get minor axis and step increment size.
    ptr = conic_list->getMinorAxisPtr();
    minor_axis = (Xgl_usgn32 *)ptr->base_ptr;
    minor_axis_step = ptr->step_size;

    //
    // Loop through the list of ellipses.
    //
    for (long i = 0; i < num_ells; i++) {
        if (XGLI_EQUAL_ZERO(*rot_angle,
            XGLI_ANGULAR_TOLERANCE)) {
            send_to_hardware_x(center->x - (*major_axis));
            send_to_hardware_y(center->y - (*minor_axis));
            send_to_hardware_w(2 * (*major_axis));
            send_to_hardware_h(2 * (*minor_axis));
        }
        else {
            send_to_hardware_x(center->x - (*minor_axis));
            send_to_hardware_y(center->y - (*major_axis));
            send_to_hardware_w(2 * (*minor_axis));
            send_to_hardware_h(2 * (*major_axis));
        }
        XGLI_INCR(center, Xgl_pt_flag_f2d*, center_step);
        XGLI_INCR(major_axis, Xgl_usgn32*, major_axis_step);
        XGLI_INCR(minor_axis, Xgl_usgn32*, major_axis_step);
        XGLI_INCR(rot_angle, float*, rot_angle_step);
    }
}

```

## *Pipeline Interfaces to XglConicData and XglRectData*

The following functions are provided by the XglConicData2d, XglConicData3d, XglRectData2d, and XglRectData3d classes. These interfaces enable the device pipeline to retrieve conic and rectangle level data for the current level or for a different level. Table 4-3 lists interfaces provided by XglConicData.

*Table 4-3* XglConicData Interfaces

Function	Description
<code>getCurrentLevel()</code>	Gets the current level number. The API data is level 0.
<code>getLevelData()</code>	Gets data for a specified level.
<code>getCurrentLevelData()</code>	Gets data for the current level.

Table 4-4 lists interfaces provided by XglConicList2d.

*Table 4-4* XglConicList2d Interfaces

Function	Description
<code>getNumConics()</code> <code>setNumConics()</code>	Get or set the number of conics in this level.
<code>getConicType()</code>	Gets the conic type, which is one of XGLI_CONIC_CIRCLE or XGLI_CONIC_ARC.
<code>getConicDataType()</code>	Gets the conic data type.
<code>getBbox()</code>	Gets the bounding box enclosing all conics of this level.
<code>getCenterPtr()</code>	Gets the pointer to the list of conic centers.
<code>getFlagPtr()</code>	Gets the pointer to the list of flags.
<code>getRadiusPtr()</code>	Gets the pointer to the list of radii.
<code>getMajorAxisPtr()</code>	Gets the pointer to the list of major axes of ellipses or elliptical arcs.
<code>getMinorAxisPtr()</code>	Gets the pointer to the list of minor axes of ellipses or elliptical arcs.
<code>getRotAnglePtr()</code>	Gets the pointer to the list of rotation angles of ellipses or elliptical arcs.
<code>getStartAnglePtr()</code>	Gets the pointer to the list of start angles of arcs.

*Table 4-4* XglConicList2d Interfaces (Continued)

Function	Description
<code>getStopAnglePtr()</code>	Get the pointer to the list of stop angles of arcs.
<code>getStartPointPtr()</code>	Get the pointer to the list of start points of arcs.
<code>getStopPointPtr()</code>	Get the pointer to the list of stop points of arcs.

Table 4-5 lists interfaces provided by the XglRectData classes.

*Table 4-5* XglRectList2d and XglRectList3d

Function	Description
<code>getNumRects()</code>	Get or set the number of rectangles in this level.
<code>setNumRects()</code>	

## *Pixel Data*

PixRects are objects that provide a uniform way of accessing and managing a 2D array of pixels. PixRects are used by the XGL core for Memory Rasters, Context fill patterns, and accumulation buffers. Device pipelines use PixRects in two ways:

- If the device pipeline uses RefDpCtx for LI-3 rendering, the pipeline will use PixRects to represent the image buffer for 2D and to represent the image buffer, Z-buffer, and accumulation buffer for 3D. See “RefDpCtx” on page 348 for information on using RefDpCtx to implement LI-3 functions.
- PixRects are used as the raster image for copy buffer operations. See page 293 in Chapter 9, “Writing Loadable Interfaces” and the section beginning on page 38 in Chapter 3, “Pipeline Framework” for information on copy buffer functions.

Pixel values in PixRects are unsigned and can be 1, 4, 8, 16, 32, or 48 bits in depth. A pixel value can be specified by an (x,y) location, and you can get or set a value at that location.

## Using PixRects

XglPixRect is the base class of the hierarchy that provides methods for using PixRects. If your device's buffers are memory mappable, the XglPixRect class has several subclasses that memory-mapped frame buffers can use to declare PixRect objects. If your device is not memory mappable or if your memory-mapped device does not correspond to Sun's memory format (see the *XGL Reference Manual* page for the format of Sun Memory Rasters), you need to subclass from XglPixRect for your frame buffer. The XglPixRect class hierarchy looks like Figure 4-6.

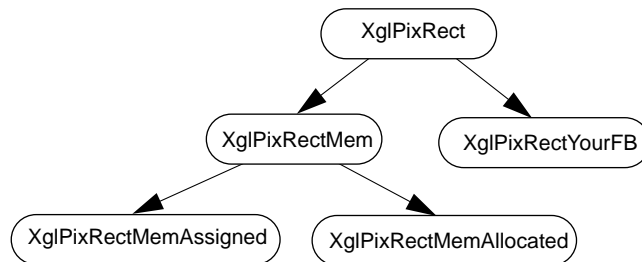


Figure 4-6 XglPixRect Class Hierarchy

## Memory-Based PixRects

The XglPixRectMem class is a specialized version of PixRect in which the underlying pixels can be addressed as memory. In this class, memory-mapped frame buffers and memory allocated via `malloc` are treated the same way. If your device is a memory-mapped frame buffer and it corresponds to the Sun memory layout, you can declare a PixRect object using one of the subclasses of XglPixRectMem.

The XglPixRectMemAssigned class sets up PixRect data structures to point to an existing piece of memory. An object of type XglPixRectMemAssigned is based on a memory-mapped frame buffer, memory allocated via `malloc`, or on an existing XglPixRectMemAllocated object. To create a PixRectMemAssigned object, declare the PixRect, allocate the memory, and assign the memory to the PixRect.

An object of type `XglPixRectMemAllocated` dynamically allocates memory itself to create a `PixRect` of a given width, height, and depth. To create an object of this type, declare the object and then call its `reallocate()` function to allocate the memory.

### *Subclassing a `PixRect` for a Non-Memory-Based Frame Buffer*

If neither the image part of the buffers nor the Z-buffer is direct memory mappable or if only one of the buffers can be accessed at a time, the device pipeline must subclass its own `PixRect` implementation from `PixRect.h`. An example of this is the case where the pixel values you want to read are not memory based but are in a register or a set of registers.

In your device `PixRect` class, you can do whatever you need to do to access the frame buffer. The `RefDpCtx` implementation requires separate `PixRect` objects for the image buffer and the Z-buffer, so you might need two objects, one for the image buffer and one for the Z-buffer, that are connected to manage the registers between them.

## *PixRect Interfaces*

Table 4-6 lists interfaces provided by `XglPixRect` and common to its subclasses. These functions describe the basic interface to a `PixRect`. Note that the color values are stored in xBGR format, where for a 24-bit RGB pixel, the physical amount of memory is actually 32 bits in which the high-order byte is unused, the next byte is blue, followed by one byte each of green and red intensity values.

*Table 4-6 XglPixRect Interfaces*

Function	Description
getValue() setValue()	Return the value of a pixel or sets the value of a pixel at the given coordinates. The PixRect must have a depth less than 32 bits, where the depth refers to the physical size rather than the layout of the pixel (in other words, a 32-bit PixRect may hold only 24 bits of information for RGB). Undefined if the coordinate values are out of bounds or the pixel is obscured. These functions must be supplied by the subclasses.
getWidth() getHeight() getDepth()	Return the size of the PixRect.
isMemory()	Returns TRUE if the PixRect can be accessed as pure memory, as when the PixRect is in memory or is a memory-mapped frame buffer, and the pixel layout corresponds to the Sun standard pixel format. See the man pages for XGL Memory Rasters for information on the Sun standard pixel format.
getWrapOriginX() getWrapOriginY() setWrapOriginX() setWrapOriginY()	Set and get wrap values are used for stipple filling where the pattern repeats itself. The origin specifies a position in the PixRect, and the get wrap value takes an (x,y) value and does a modulus operation on the value with the width and height, and returns the value at that modulus position. This is used for filling where the fill pattern is represented as a PixRect.
getWrappedValue()	Subtracts wrapOrigin from the point, wraps at the edge of the PixRect, and returns the value.
fillRectangle()	Sets a rectangular region with a given value.
getValueByPointer() setValueByPointer()	Handle very large PixRects. Specifically, these functions handle 48-bit deep PixRects, which are used by the accumulation operation.

Table 4-7 lists the interfaces provided by the XglPixRectMem class.

*Table 4-7* XglPixRectMem Interfaces

Function	Description
<code>getLineBytes()</code>	Returns the number of bytes per scan line, including any possible padding at the end of the PixRect.
<code>getMemoryAddress()</code>	Given an (x,y) location, this function returns a pointer to the address of the pixel at that location.
<code>getMemoryAddress1()</code> <code>getMemoryAddress4()</code> <code>getMemoryAddress8()</code> <code>getMemoryAddress16()</code> <code>getMemoryAddress32()</code> <code>getMemoryAddress48()</code>	Inline versions of <code>getMemoryAddress()</code> .
<code>getValue1()</code> <code>getValue4()</code> <code>getValue8()</code> <code>getValue16()</code> <code>getValue32()</code> <code>setValue1()</code> <code>setValue4()</code> <code>setValue8()</code> <code>setValue16()</code> <code>setValue32()</code>	Inline versions of <code>getValue()</code> and <code>setValue()</code> .

Table 4-8 lists the interfaces provided by the XglPixRectMemAllocated class.

*Table 4-8* XglPixRectMemAllocated Interfaces

Function	Description
<code>realloc()</code>	Returns the address of the newly allocated memory raster. NULL if allocation fails.
<code>dealloc()</code>	Frees memory used for the PixRect.

Table 4-9 lists the interfaces provided by XglPixRectMemAssigned.


*Table 4-9* XglPixRectMemAssigned Interfaces

Function	Description
<code>reassign()</code>	Creates a PixRect on existing memory.



## *Handling Changes to Object State*

---

5 

This chapter describes how a device pipeline gets information about changes to XGL state. The chapter includes information on the following topics:

- Changes to Context state and changes to objects associated with the Context
- Changes to Device state
- Design issues to think about when implementing state handling in a device pipeline



As you read this chapter, you will find it helpful to have access to the header files for the stroke groups and the header file defining the messages. These are:

- `StrokeGroup.h` and `StrokeGroup3d.h`
- `Msg.h`

## *State Changes and the Device Pipeline*

The device pipelines are notified directly of Context attribute changes by the `objectSet()` and `messageReceive()` functions through the `opsVec[]` function array. Information about XGL object state is contained in the following device-independent objects:

- Context and Context 3D objects store information about Context state.
- Stroke group objects store information about certain multipolyline attributes and store the attribute value for these attributes as well.
- The Context's view cache object stores information about items derived from the Context's view model attributes.
- The set of Device objects store information about Device state.

The process of determining what attributes have changed and getting updated attribute values is described in the following sections.

## *Getting Attribute Values from the Context Object*

The device pipelines are notified of Context attribute changes as soon as the application sets new attributes on the Context. Each `xgl_object_set()` calls the pipeline version of `objectSet()` through the `XGLI_LI_OBJ_SET` entry of the `opsVec[]` array. When the application calls the API object set function, the following events occur:

1. The device-independent wrapper updates the Context object, and the entire list of attribute types is processed.
2. The current device pipeline is determined from the Context object.
3. The wrapper calls the pipeline `XglDpCtx opsVec[XGLI_LI_OBJ_SET]` entry with the list of attribute types. The attribute values are not passed to the pipeline in this list.

The pipeline `objectSet()` function gets the NULL-terminated list of attribute types forwarded by the device-independent core. To provide an `objectSet()` function, declare the function as a member of your pipeline `XglDpCtx`, and in the `XglDpCtx`'s constructor, put a pointer to the function in the `XGLI_LI_OBJ_SET` entry of the `opsVec[]` array, as follows:

```
opsVec[XGLI_LI_OBJ_SET] = XGLI_OPS(XglDpCtx3dFb::objectSet);
```

The `objectSet()` function is called when the application sets a Context attribute value. This function will provide a switch statement for the attributes the pipeline is interested in. The switch statement can ignore attribute types that the pipeline isn't interested in and can combine attribute types that can be handled in the same way. The pipeline can either update the hardware context immediately from within `objectSet()` or note that changes have occurred and update the hardware at a later time.

The following sample code shows a pipeline `objectSet()`:

```
//
// Example for the generic DP set function
//
XglDpCtx3dGx::objectSet(const Xgl_attribute *attr_type)
{
    for(; *attr_type; attr_type++) {
        switch (*attr_type) {

            case XGL_CTX_LINE_COLOR: // set line color in DP
            {
                Xgl_color *line_color =
                    ctx->getCurrentStroke()->getColor();
                // send line color to hardware
                ...
            }
            break;

            case ATTR_A:      // combine attributes
            case ATTR_B:
            case ATTR_C:
                do_something();
                ...

            case default:
                // ignore attribute
                break;

        }
    }
}
```

See `Context.h` for the Context interfaces the pipeline can use to get Context attribute values. Note that if the device pipeline does not implement `objectSet()`, it will have to check the Context attributes at rendering time.

This might be an appropriate design for an LI-3 pipeline that is concerned with a small subset of attributes. However, implementing `objectSet()` is advisable for most pipelines for performance reasons.

## *When the Device Associated with a Context Is Changed*

When the device-independent code calls the device pipeline's `objectSet()` to connect a Context to a Device, the device pipeline will receive only `XGL_CTX_DEVICE` in the attribute list. In this case, it is the device pipeline's responsibility to update all concerned Context attributes. To do this, the device pipeline can use the Context utility function `getAttrTypeListAll()`, which returns a pointer to a static list of all XGL Context attributes. The Context attribute list contains both 2D and 3D attributes.

An example of how a device pipeline can handle the `objectSet()` case for `XGL_CTX_DEVICE` is shown in the code fragment below.

```
XglDpCtx3dGx::objectSet(const Xgl_attribute *attr_type)
{
    for(; *attr_type; attr_type++) {
        switch (*attr_type) {

            case XGL_CTX_DEVICE: // new context attached
                objectSet(ctx->getAttrTypeListAll());
                break;

            case...
                ...
        }
    }
}
```

Since `getAttrTypeListAll()` returns a list of all 2D and 3D Context attributes, it is recommended that the device pipeline create its own separate 2D and 3D Context attribute lists for optimum performance. The device pipeline could create static lists in its `XglDpCtx[2,3]d` pipeline classes.

If `XGL_CTX_DEVICE` is embedded in an `xgl_object_set()` call, as shown in the API call below, all Context attributes are updated with the API data included in the call before the device pipeline `objectSet()` is called. Thus, all device-independent Context attributes are up-to-date when a device pipeline receives `XGL_CTX_DEVICE`.

```
xgl_object_set(ctx, XGL_CTX_LINE_COLOR, my_line_color,  
               XGL_CTX_DEVICE, my_ras,  
               XGL_CTX_NEW_FRAME_ACTION, my_new_frame_action,  
               XGL_CTX_PLANE_MASK, -1,  
               0);
```

## *Getting Attribute Values from Objects Other Than the Context*

The device pipeline is notified immediately of changes to objects other than the Context by the message passing mechanism. In XGL, when objects are instantiated, other objects can register interest in the new objects and become users of the objects. During program execution, when the used object's attributes change, the object sends a message to its users informing them of the change. For example, the Context becomes a user of the Line Pattern, Stroke Font, and Marker objects. When the Line Pattern changes, it sends a message about the change to the Context. When the Context receives an object message,, it updates its data and forwards the message to the device pipeline by calling the `XglDpCtx messageReceive()` function through the `opsVec[]` `XGLI_LI_MSG_RCV` entry.

The pipeline `messageReceive()` function gets a pointer to an XGL object type and a message of type `XglMsg`. To provide a `messageReceive()` function, declare the function as a member of your pipeline `XglDpCtx`, and in the `XglDpCtx`'s constructor, put a pointer to the function in the `XGLI_LI_MSG_RCV` entry of the `opsVec[]` array, as follows:

```
opsVec[XGLI_LI_MSG_RCV] = XGLI_OPS(XglDpCtx3dFb::messageReceive);
```

When you have done this, the `messageReceive()` function will always be called when there is a message for the `XglDpCtx`. The `messageReceive()` function will check the object type and message, and respond appropriately.

The pipeline can use the `messageReceive()` function to adjust to object changes. For example, if the hardware caches colors, the `XglDpCtx` can update the cached colors when `messageReceive()` receives a message that the color map changed. If the device caches a line pattern or light in its hardware, a

message about these objects indicates that the hardware context may need updating. The function can ignore messages that the pipeline is not concerned with.

The objects and messages are listed in Table 5-1. The default message, `XGLI_MSG_STANDARD`, simply indicates that an object has changed; it does not provide information about what changed or about what attribute caused the change. For the standard message type, the device pipeline can check individual attributes relevant to the object or reload the entire object into the hardware. See page 105 for more information on using the view group messages.

*Table 5-1* Object Messages

Object-Message	Description
<code>XGL_2D_CTX / XGL_3D_CTX</code>	
<code>XGLI_MSG_VIEW_COORD_SYS</code>	View group coordinate system changed, or push or pop of the current coordinate system. You should check derived data.
<code>XGLI_MSG_VIEW_CTX_ATTR</code>	View group Context attribute changes. This message corresponds to an API attribute that modifies derived data. You should check derived data.
<code>XGL_LIGHT</code>	
<code>XGLI_MSG_STANDARD</code>	The Light object has changed. You may need to check derived data. Update cached information regarding this Context attribute. <code>XGL_3D_CTX_LIGHT</code>
<code>XGL_LPAT</code>	
<code>XGLI_MSG_STANDARD</code>	The line pattern or edge pattern has changed. Update cached information regarding these Context attributes. <code>XGL_CTX_LINE_PATTERN</code> <code>XGL_CTX_EDGE_PATTERN</code>
<code>XGL_MARKER</code>	
<code>XGLI_MSG_STANDARD</code>	The user-defined marker has changed. Update cached information regarding this Context attribute. <code>XGL_CTX_MARKER</code>

---

Table 5-1 Object Messages

Object-Message	Description
XGL_MEM_RAS XGLI_MSG_STANDARD	Front or back surface fill pattern memory raster has changed. Update cached information regarding these Context attributes. XGL_CTX_RASTER_FPAT XGL_CTX_SURF_FRONT_FPAT XGL_3D_CTX_SURF_BACK_FPAT
XGL_SFONT_n XGLI_MSG_STANDARD	Stroke Font object has changed. Update cached information regarding this Context attribute. XGL_CTX_SFONT_n
XGL_TMAP XGLI_MSG_STANDARD	Front or back Texture Map object has changed. Update cached information regarding these Context attributes. XGL_3D_CTX_SURF_FRONT_TMAP XGL_3D_CTX_SURF_BACK_TMAP
XGLI_MSG_TEXTURE_DESC	Texture Map descriptor has changed, and possibly the MipMap Texture object has changed. You may need to recache the MIP map.
XGL_TRANS XGLI_MSG_STANDARD	Global model transform, local model transform, or view transform has changed. Check derived data. Update cached information regarding these Context attributes. XGL_CTX_GLOBAL_TRANS XGL_CTX_LOCAL_MODEL_TRANS XGL_CTX_VIEW_TRANS In the 3D Context, the normal transform has changed. Check this attribute. XGL_3D_CTX_NORMAL_TRANS
XGL_WIN_RAS XGLI_MSG_DEV_MULTIBUFFER	Multibuffering has been set on the device. Update cached information regarding this Raster attribute. XGL_WIN_RAS_MULTIBUFFER

Table 5-1 Object Messages

Object-Message	Description
XGL_WIN_RAS / XGL_MEM_RAS XGLI_MSG_DEV_COLOR	Color map has or depth pixel mapping has changed. Update cached information regarding these Context attributes. XGL_CTX_PLANE_MASK XGL_CTX_MARKER_COLOR XGL_CTX_LINE_COLOR XGL_CTX_LINE_ALT_COLOR XGL_CTX_SURF_FRONT_COLOR XGL_CTX_SURF_BACK_COLOR XGL_CTX_BACKGROUND_COLOR Note that a message is sent both when the device is assigned a new color map and when a change is made to an existing color map.
XGLI_MSG_DEV_DIM	Window width or height has changed, window raster has been resized, or rect list has changed. You should check derived data.
XGLI_MSG_DEV_OTHER	Changes in image buffer address, Z buffer address, source buffer, buffer display, buffer draw, buffer minimum delay, double buffer draw, number of buffers allocated, stereo mode. See the man pages for the Device attributes.
XGLI_MSG_RAS_CLIP	Rect list has changed. Update cached information regarding this Raster attribute. XGL_RAS_RECT_LIST

The following sample code shows a pipeline `messageReceive()` function. This routine notes object changes, creates an attribute type list with attributes it is interested in, and sends the attribute type list to the pipeline `objectSet()` function. The pipeline `objectSet()` updates the hardware. This routine also sets a `transformChanged` flag so that it can point the `opsVec` renderer to the generic renderer. The generic renderer will check the `transformChanged` flag, and, if necessary, it will check the view group using `viewGrpItf->changedComposite(viewConcern)` to see what changed in derived data. An alternate design for `messageReceive()` would be to update the hardware from within the function.



```
void XglDpCtx2dGx::messageReceive(XglObject* obj,
                                   const XglMsg& msg)
{
    switch (obj->getObjType()) {

        case XGL_2D_CTX:
        case XGL_3D_CTX:
            if (msg.flag & (XGLI_MSG_VIEW_COORD_SYS |
                           XGLI_MSG_VIEW_CTX_ATTR)) {
                transformChanged = TRUE;
                // Set generic renderers.
            }
            break;

        case XGL_WIN_RAS:
            if (obj == device) {
                if (msg.flag & XGLI_MSG_DEV_COLOR) {
                    //
                    // Update cached colors and plane mask changes.
                    //
                    attrTypeList[0] = XGL_CTX_MARKER_COLOR;
                    attrTypeList[1] = XGL_CTX_LINE_COLOR;
                    attrTypeList[2] = XGL_CTX_LINE_ALT_COLOR;
                    attrTypeList[3] = XGL_CTX_SURF_FRONT_COLOR;
                    attrTypeList[4] = XGL_CTX_BACKGROUND_COLOR;
                    attrTypeList[5] = XGL_CTX_PLANE_MASK;
                    attrTypeList[6] = XGL_UNUSED;
                    objectSet((const Xgl_attribute*) attrTypeList);
                }

                if (msg.flag & XGLI_MSG_DEV_DIM) {
                    transformChanged = TRUE;
                    // Set generic renderers.
                }

                if (msg.flag & XGLI_MSG_DEV_OTHER) {
                    // Re-evaluate the number of buffers to render to
                    // based on bufferAllocated and XGL_CTX_RENDER_BUFFER

                    attrTypeList[0] = XGL_CTX_RENDER_BUFFER;
                    attrTypeList[1] = XGL_UNUSED;
                    objectSet((const Xgl_attribute*) attrTypeList);
                }
            }
    }
}
```

```

        // Set generic renderers.
    }
}
break;

case XGL_LPAT:
    if (obj == ctx->getLinePattern() ||
        obj == ctx->getEdgePattern()) {
        attrTypeList[0] = XGL_CTX_LINE_PATTERN;
        attrTypeList[1] = XGL_UNUSED;
        objectSet((const Xgl_attribute*) attrTypeList);
    }
    break;

case XGL_MARKER:
    if (obj == (XglObject*)ctx->getMarker()) {
        attrTypeList[0] = XGL_CTX_MARKER;
        attrTypeList[1] = XGL_UNUSED;
        objectSet((const Xgl_attribute*) attrTypeList);
    }
    break;

case XGL_TRANS:
    if (obj == (XglObject*)ctx->getGlobalModelTrans() ||
        obj == (XglObject*)ctx->getLocalModelTrans() ||
        obj == (XglObject*)ctx->getViewTrans()) {

        transformChanged = TRUE;
        // Set generic renderers.
    }
    break;
}

```

For additional information on object relationships, see the *XGL Architecture Guide*.

### *More on Device State Changes*

In addition to passing device state changes to the pipeline via the message passing mechanism, the device-independent code notifies the device pipeline of device changes by calling the `set . . . ()` functions defined in the `XglDpDev` class hierarchy. The `XglDpDev` functions enable the device to make device-

specific changes. For example, in addition to the color map change message that is sent to the XglDpCtx, there are two virtual XglDpDev functions that are called when the color map or pixel mapping changes.

```
virtual void    setCmap(XglCmap*);
virtual void    setPixelMapping(const Xgl_usgn32[]);
```

See page 52 for information on the XglDpDev optional functions.

## *Handling Derived Data Changes*

While `objectSet()` and `messageReceive()` enable the pipeline to keep track of the state of API attributes, the derived data facility is used to maintain data that are derived from the API attributes. The device pipeline is notified when derived data (view interface) changes by the message mechanism. When a derived data change occurs, the view object sends a message to the device pipeline by calling the `XGLI_LI_MSG_RCV` entry of the `opsVec[]` array, with the Context as the object type, and a message bitfield indicating what in derived data has changed.

Derived data changes that generate messages are set coordinate system, pop coordinate system, and any API Context attribute that could affect derived data. Device pipeline implementers may want to ignore the message flags and use `checkchangedComposite()` to see if any updating needs to be done whenever they receive a Context object message. The message flags for derived data changes are shown in Table 5-1 on page 100.

The device pipeline will set the view interface message in its `messageReceive()` function when the object type is `XGL_2D_CTX` or `XGL_3D_CTX`. Some example code to do this is shown below.

```
// DP's receive message function. XGLI_LI_MSG_RCV slot
// in ops vector points to this:

void XglDpCtx2dGX::messageReceive(XglObject *obj,
                                   const XglMsg& msg)
{
    switch (obj->getObjType()) {
        XGL_2D_CTX:
            if (msg.flag & (XGLI_MSG_VIEW_CTX_ATTR |
                           XGLI_MSG_VIEW_COORD_SYS)) {
                // update DP's derved data,
                // check viewGrpItf->changedComposite(<view concerns>)
```

```
// DP's receive message function. XGLI_LI_MSG_RCV slot
    }
    break;

    ... //other message processing
}
}
```

Information about derived data changes is computed in a lazy manner at a pipeline's request. See Chapter 7, "View Model Derived Data" for information on derived data.

## *Getting Stroke Attribute Values from the Stroke Group Object*

The stroke group is the source from which the pipeline obtains the values for the line attributes, such as line color, during a `multiPolyline()` call. The idea behind the stroke group is to make the drawing of different strokes types as transparent as possible to a device pipeline that doesn't support all stroke primitives.

The primitives that may be rendered as multipolylines are lines, markers, text, edges, and hollow polygons. These primitives are considered to be *stroke types*. Since the same set of attributes (but different attribute values) applies to each of the stroke types when rendered as lines, the Context object maintains a stroke group object for each of the stroke types. For 2D, the stroke groups are line, marker, text, edge, and front surface. For 3D, the stroke groups are the 2D groups and the back surface group.

The stroke group object contains the actual attribute values for the stroke attributes. The stroke attributes are:

- Antialiasing blend equation
- Antialiasing filter width
- Antialiasing filter shape
- Alternate color
- Cap
- Color
- Color selector
- Join
- Miter limit
- Pattern
- Style

- Width scale factor
- Flag mask
- Expected flag mask

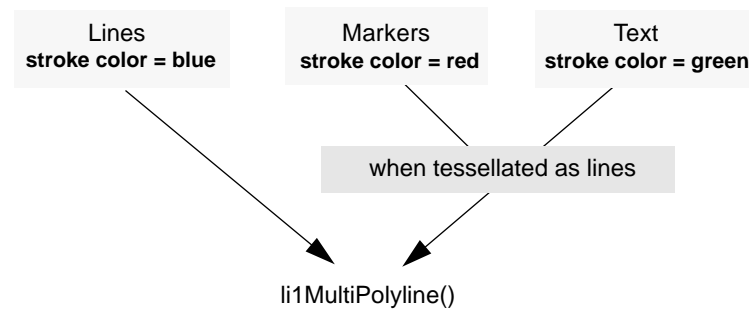
Most of the stroke attributes map to API attributes. However, flag mask and expected flag mask in `StrokeGroup.h` are specific to the stroke group object and depend on the stroke type; they are not API attributes and have no corresponding attribute type. For 3D rendering, the stroke group object is extended to include values for color interpolation and DC offset. Like flag mask and expected flag mask, DC offset in `StrokeGroup3d.h` does not map directly to an API attribute. See page 111 for information on flag mask and DC offset.

### *Example of Device Pipeline Use of Stroke Groups*

Let's consider a device pipeline that cannot render text in hardware. In this situation, the text primitive will go through the software pipeline where it is tessellated into polylines. Before the polylines are handed to the device pipeline's LI-1 polyline renderer, the Context is told to activate the text stroke group. This activation sets the current stroke to the text stroke group and informs the device pipeline which stroke attributes have changed<sup>1</sup>. When the device pipeline reads the changed attributes out of the current stroke group, it gets the text attributes. For example, if the current line color is blue but the text color is green, the device pipeline will get the color green from the current stroke group. Figure 5-1 on page 108 illustrates this concept.

---

1. Only attributes that have actually changed between the old and new stroke groups will be sent to the device pipeline. For example, if the old stroke group was polylines and the line width was 1.0, changing the stroke group to text (whose line width is always 1.0) will not cause the line width attribute to be sent to the device pipeline.



Case 1: if text is rendered → multipolylines are rendered in green

Case 2: if lines are rendered → multipolylines are rendered in blue

*Figure 5-1* Attribute Processing Using the Stroke Group

As long as text continues to be rendered, the text stroke group will remain the current stroke group. The current stroke group will change when either polylines are rendered, or another non-text stroke primitive falls back to the software pipeline for rendering.

Changes to stroke attributes are transmitted directly to the device pipeline. The only difference is that the device pipeline will see twice as many stroke attributes when anything other than the polyline stroke group is active. Thus, a device pipeline that fully accelerates text at LI-1 would see that `XGL_CTX_STEXT_COLOR` has changed, and a device pipeline that does not accelerate text at LI-1 would see that both `XGL_CTX_STEXT_COLOR` and `XGL_CTX_LINE_COLOR` have changed. It is necessary to pass the `XGL_CTX_STEXT_COLOR` attribute so that device pipelines which support text at LI-1 in some circumstances have a consistent view of the Context state.

See `StrokeGroup.h` and `StrokeGroup3d.h` for the interfaces a pipeline uses to obtain attribute values from the stroke group.

## *Rendering Multipolylines*

Rendering polylines involves getting a pointer to the current stroke group and obtaining the attribute values that have changed from the stroke group. To indicate which stroke group will be used for rendering, the Context object provides a current stroke pointer that points to one of the stroke group objects. When the device pipeline receives a request to render a multipolyline, it gets the pointer to the current stroke group using the Context interface

```
getCurrentStroke():
```

```
cur_stroke = ctx->getCurrentStroke()
```

### *Procedure for Getting Attribute Values for `xgl_multipolyline()`*

For most primitives, new attribute values are obtained from the Context object. However, the difference between the attribute processing for an `xgl_multipolyline()` call and for other primitive rendering calls is that the values for the stroke attributes are obtained from the stroke group pointed to by the Context's current stroke pointer.

The steps for obtaining attribute values when rendering multipolylines are listed below.

1. The pipeline gets the current stroke pointer using the Context interface  

```
cur_stroke = ctx->getCurrentStroke();
```
2. The pipeline obtains the attribute values from the stroke group for changes in the line attributes. To get the line color, for example, the pipeline requests the line color with `cur_stroke->getColor()`. Values for attributes not in the stroke group are obtained from the Context, as in  

```
ctx->getDepthCueMode();
```
3. The pipeline loads the new values into hardware.

---

**Note** – The stroke group is designed to hide the actual type of stroke it is rendering from the pipeline. Normally, a device pipeline should get line group attributes from the `XglStrokeGroup` object for all multipolyline rendering unless the device pipeline can accelerate all primitives completely at LI-1 and will never call the software pipeline for tessellation. If a device pipeline does accelerate a stroke primitive (for example, it implements `li1StrokeText()`), the device pipeline can obtain the text attributes from the Context rather than from the stroke group. If you are absolutely sure that your pipeline does not fall back on the software pipeline for any of the stroke primitives (edges, text, markers, and hollow polygons) and that there is no chance of the stroke group being anything other than `lineStrokeGroup`, then your pipeline can get line group attributes directly from the Context. For primitives other than `multiPolyline()` that depend on the line attributes, the values for the line changes can be retrieved from the Context.

---

### *Procedure for Getting Attribute Values That Have Changed*

The `assignCurStrokeAs<prim>()` functions are used by the software pipeline to change the current stroke group, and to call the device pipeline `objectSet()` function to inform the pipeline that certain stroke group attributes have changed.

Currently, when `assignCurStrokeAs<prim>()` is called, the device pipeline `objectSet()` function is also called notifying the pipeline of all changed line attributes. This means the device pipeline should load the current stroke attribute list for lines.

These `objectSet()` calls occurs in two different circumstances.

- `assignCurStrokeAs<prim>()` is called by pipelines, changing the current stroke group (`currentStrokeGroup`).
- Attributes corresponding to the current stroke group setting are changed by an `objectSet()` call. For example, if a text attribute, such as text color, changes while the current stroke group is pointing to the text group, `objectSet()` will be called after calling the software pipeline. This will call `objectSet()` through the `XGLI_LI_OBJ_SET` entry of the `opsVec[]` array sending the changed text attribute(s) as line attributes. In this



scenario, the device pipeline would receive an `objectSet()` call with a list of attribute types sent from the API, indicating which stroke attributes must be updated from the strokeGroup object.

If the device pipeline never needs stroke groups, it can process all the attributes directly from the Context and ignore the stroke group object. An intermediate approach is possible, since stroke groups only happen if the device pipeline calls the software pipeline for a particular case (stroke text, for example). In these cases, only the stroke groups still used by the device pipeline will generate an `objectSet()`.

### *Flag Mask and Expected Flag Value*

In XGL an application can provide flag information at each point of a primitive. This flag information determines whether specific line segments within the polyline are drawn. The stroke group flag mask and expected flag mask attributes are useful when the point type of the multipolyline being rendered has flag information.

If the point type has flag information, the pipeline ANDs the flag information in the vertex data with the `flagMask` from the stroke group and compares it to the `expectedFlagValue` from the stroke group. If they are equal, the line should be drawn; otherwise, the line should not be drawn.

Table 5-2 shows the flag information for the different stroke types.

*Table 5-2* Stroke Table Flag Mask and Expected Flag Mask Values

Stroke Group	Flag Mask	Expected Flag Mask
Line stroke group	XGL_DRAW_EDGE	XGL_DRAW_EDGE
Edge stroke group	XGL_DRAW_EDGE   XGL_EDGE_IS_INTERNAL	XGL_DRAW_EDGE
Marker stroke group	No bits set	No bits set
Front surface stroke group	XGL_EDGE_IS_INTERNAL	No bits set
Back surface stroke group	XGL_EDGE_IS_INTERNAL	No bits set
Text stroke group	No bits set	No bits set

For example, the flag information for lines is `XGL_DRAW_EDGE`, whereas the flag information for edges could be `XGL_DRAW_EDGE` and/or `XGL_EDGE_IS_INTERNAL`. In the case of lines, the pipeline needs to determine whether `XGL_DRAW_EDGE` is set in the flag information before rendering the line. In the case of edges, the pipeline draws the edge when the `XGL_DRAW_EDGE` bit is set but not when `XGL_EDGE_INTERNAL` is set. Thus, when different stroke types are rendered as lines, the stroke group object provides `getFlagMask()` and `getExpectedFlagValue()` to make the dissimilarity in flags transparent to the device pipeline.

Example pseudocode to use these flags might be:

```
Xgl_pt_flag_f3d    pt;
if (pt_type has flag) {
    if ((pt.flag & cur_stroke->getFlagMask()) ==
        cur_stroke->getExpectedFlagValue()) {
        // Draw the line
    }
}
```

---

**Note** – At LI-1, since the point type can have flag data only when rendering lines (text and markers when rendered as lines cannot have point type with flag data), it is correct to assume that `flagMask` and `expectedFlagValue` are always the same (`XGL_DRAW_EDGE`) for `li1MultiPolyline()`.

---

## *DC Offset*

Some stroke types need to have the Z value adjusted either to ensure visual correctness or to respond to the setting of the API attribute `XGL_3D_CTX_SURF_DC_OFFSET`. The DC offset attribute is provided in `StrokeGroup3d.h` so that this is handled by the device pipeline. It determines if the Z of a line should be closer, unchanged, or farther than the original Z value of the line. The DC offset attribute can take on these enumerated values:

- `XGLI_DC_OFFSET_NONE` – The DC offset attribute is set to this value for the line, marker, and text stroke groups. The pipeline does not need to adjust the Z value.

- `XGLI_DC_OFFSET_FRONT` – The DC offset is set to this value when rendering edges as lines. It ensures that the edges appear on top of the polygon. The pipeline should subtract an offset from the Z component of each vertex of the multipolyline so that the line appears to be in front.
- `XGLI_DC_OFFSET_BACK` – Used when hollow polygons are drawn as lines. This maps to the XGL API attribute `XGL_3D_CTX_SURF_DC_OFFSET`.

The DC offset values for the stroke groups are listed in Table 5-3.

*Table 5-3* Stroke Group DC Offset Values

Stroke Group	DC Offset Value
Line stroke group	<code>XGLI_DC_OFFSET_NONE</code>
Edge stroke group	<code>XGLI_DC_OFFSET_FRONT</code>
Marker stroke group	<code>XGLI_DC_OFFSET_NONE</code>
Front surface stroke group	<code>XGLI_DC_OFFSET_BACK</code> if <code>XGL_3D_CTX_SURF_DC_OFFSET</code> is <code>TRUE</code> <code>XGLI_DC_OFFSET_NONE</code> if <code>XGL_3D_CTX_SURF_DC_OFFSET</code> is <code>FALSE</code>
Back surface stroke group	<code>XGLI_DC_OFFSET_NONE</code>
Text stroke group	<code>XGLI_DC_OFFSET_NONE</code>

Thus, a pipeline adjusts the Z value according to value returned by the `getDcOffset()` function in the stroke group object. Note that the DC offset attribute is relevant only when Z-buffering is enabled.

**Note** – The software pipeline does not set the current stroke group to the edge stroke group, front-surface stroke group, or back-surface stroke group at the LI-1 layer. But, if a device falls back to software for text and markers, the current stroke can be either text/markers or line. But since the DC offset is not used by text/markers or line stroke groups, you can ignore the DC offset at `lilMultiPolyline()`.

## *Design Issues*

There are several issues that you may want to consider when implementing the processing of state changes. At the most elementary level, a pipeline must do the following when rendering a primitive:

1. Decide whether it can render the primitive and call the software pipeline if it cannot render the primitive.
2. Map XGL Context state into the hardware state.
3. Render.

Within this scheme, the pipeline must be able to handle changes in Context state and API data, changes in the LI level of the primitive, and changes in the XGL Context being used to render.

### *Deciding to Reject a Primitive*

The decision on whether the pipeline can render a primitive depends in part on the values of the attributes for the primitive. This means that the pipeline must process state information before it can conclude whether it can render the primitive.

A pipeline has two choices when evaluating attributes. It can abort processing if it finds an attribute it cannot accelerate (for example, if line width is greater than some value) or if the API information cannot be accelerated (for example, if the point type is homogeneous). The code fragment below shows an example of a pipeline calling the software pipeline to process wide lines.

```
void DpCtx3dExampleDp::lilMultiPolyline(Xgl_bbox*    api_bbox,
                                         Xgl_usgn32  api_num_plists,
                                         Xgl_pt_list* api_pt_list)
{
    const XglStrokeGroup3d* cur_stroke = ctx->getCurrentStroke();
    if (cur_stroke->getWidthScaleFactor() >= 2.0) {
        swp->lilMultiPolyline(api_bbox, api_num_plists,
                              api_pt_list);
        return;
    }
    // Continue with lilMultiPolyline...
}
```

## *Handling Context Switches*

An application may be using any number of XGL Contexts. For example, it may use a different Context for each view of the geometry that it wants to display, or it may use different Contexts for areas of the window. It is the responsibility of the pipeline to update hardware state when the application switches the XGL Context that it is using to render.

One way to implement this might be to check which Context is being used to render when the primitive is entered. If the current Context is the same as the last Context, the function can continue other processing. If the Context is different from the last Context used, then the function should assume that the hardware state is invalid and take appropriate action. For example:

```
if (dp_last_xgl_ctx != ctx) {  
    //  
    // Update derived data.  
    //  
    viewGrpItf->setComposite();  
  
    //  
    // Update all context attributes.  
    //  
    // All relevant attributes must be updated. The device  
    // pipelines objectSet() may be used.  
    //  
    objectSet(ctx->getAttrTypeListAll()); // DI utility list.  
  
    dp_last_xgl_ctx = ctx;  
}
```

How you handle updating your hardware after Context switches is an implementation decision left to you. Note that you may have to invalidate your hardware state when the Context changes only if you map all XGL Contexts to a single hardware context.

### *Handling Changes in LI Levels*

The way that a pipeline processes state information must be handled carefully to account for switching of loadable interface levels. The possible problem that you might want to consider is at different loadable interface levels, the list of attributes that the pipeline must handle is not the same. This problem occurs when the device pipeline rejects a primitive outright at LI-1 (as when it hasn't implemented the primitive at the LI-1 level) but is called to render at the LI-2 or LI-3 level. For example, consider the case when, before a multipolyline call, the user changes both the line width and the line color. Assume also that the user sets the line width to be greater than 1 (wide line). The device pipeline cannot handle wide lines, so it falls back to the software pipeline at LI-2 and renders the wide lines at LI-3. At LI-3, the pipeline gets the value of the line color.

### *Partial Rendering of a Primitive*

For the case in which a device pipeline calls the software pipeline to render some of the primitive's geometry and continues processing the rest of the primitive on its own, it is the device pipeline's responsibility to restore the hardware to the correct state before rendering the rest of the primitive. In other words, during the time that the software pipeline is processing its part of the geometry, the state of the hardware may change, and the device pipeline cannot rely on `objectSet()` to notify the pipeline of this change.


For example, during a `multiSimplePolygon()` call, if a device pipeline cannot render a complex polygon, it calls the software pipeline. At LI-2 or LI-3, the device pipeline must disable some attributes, such as model clipping or MC to DC transformations, which are already done by the software pipeline at LI-1. When the control returns to the device pipeline to render the remaining simple polygons, the device pipeline may need to set up the hardware to render the polygons at LI-1 because the state of the hardware has changed. The device pipeline now needs the hardware to do model clipping and other operations, and has to set up the hardware accordingly.





## *Getting Information from XGL Objects*

---

6 

This chapter describes how a device pipeline gets information from XGL objects and uses object interfaces. The chapter includes information on the following topics:

- Getting information from the Context and from objects associated with the Context
- Getting information from the Device and Color Map



As you read this chapter, you will find it helpful to have access to the following header files:

- `Context.h`, `Context2d.h`, and `Context3d.h`
- `Cmap.h`
- `Device.h`, `Raster.h`, `RasterWin.h`, and `RasterMem.h`
- `DmapTexture.h`
- `Light.h`
- `LinePattern.h`
- `Marker.h`
- `MipMapTexture.h`
- `Sfont.h`
- `Tmap.h`
- `Transform.h`

## What You Should Know About XGL Attribute Values

The values of XGL attributes are stored in the Context object and in API objects associated with the Context object, such as the Light object and the Transform object. At rendering time, the device pipeline often needs to get information on various attributes from within its XglDpCtx and XglDpDev objects.

The pipeline is linked to a specific Context and a specific Device through its XglDpCtx object; from the XglDpCtx, the pipeline can get to the Context attributes it needs and to the attributes of objects associated with the Context. Similarly, the pipeline is linked to the Device object through its XglDpDev object, and it can get information on the device-independent Device object and the Color Map object through the XglDpDev. Figure 6-1 shows the architecture of the device-independent objects and their relationship to pipeline objects. In this illustration, the filled arrows denote a permanent relationship; for example, the XglDpCtx object is always linked to a unique Context object and unique Device object. The unfilled arrows show possibly transitory API relationships.

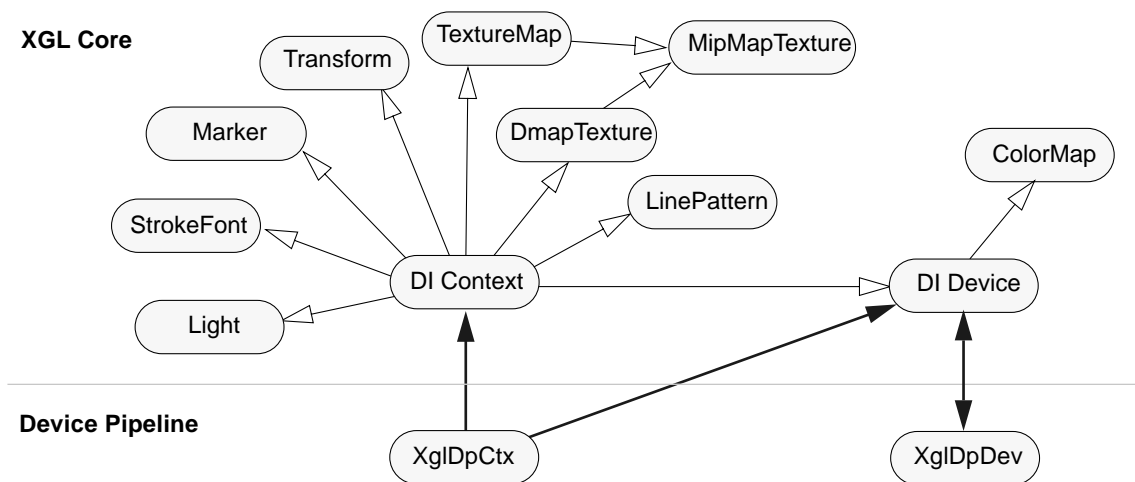


Figure 6-1 DI and Dp Object Relationships

Pipelines access API attribute data via public methods in the public interface of the API object classes; the data itself is not exposed in the public interface or accessible to the pipeline. To see the interface for an object, look at the class hierarchy for the object.

Part of the public interface implements the XGL API. Thus, in the API object classes, there are two categories of functions: functions that correspond closely to API attributes and other functions that are for internal uses, including the device pipeline (flagged with `XGL_INTERNAL`). A third category is reserved for the XGL core and is inaccessible to the device pipelines (flagged with `XGL_CORE`). In the public interface, you will notice a number of `set...()` functions; for the most part, these implement the API set functions and are not meant to be used by the pipeline. An exception to this is the pipeline use of `device->setBufDisplay()` and `device->setBufDraw()` from within its `XglDpCtx lilNewFrame()` primitive.

### *Naming Conventions for Internal Attributes*

The mapping of an API attribute name to its corresponding C++ method is handled in a standard way. For example, in `Light.h`, you will see the function `getColor()`. This function gets the light color and corresponds to the API attribute `XGL_LIGHT_COLOR`. The naming conventions for internal attributes, such as a hypothetical API attribute `XGL_CLASS_ATTRIBUTE_HAS_WORDS`, are as follows:

- The internal method to get the attribute is `getAttributeHasWords()`.
- The method is declared in the `XglClass` class in the `Class.h` header file.

Here are some examples:

- For the Context attribute `XGL_CTX_MARKER_COLOR`, the function `getMarkerColor()` is declared in the `XglContext` class in `Context.h`.
- For the Context attribute `XGL_3D_CTX_SURF_FRONT_ILLUMINATION`, the function `getSurfFrontIllumination()` is declared in the `XglContext3d` class in `Context3d.h`.
- For the Device attribute `XGL_DEV_COLOR_MAP`, the function `getCmap()` is declared in the `XglDevice` class in `Device.h`.

---

**Note** – In some cases, although an attribute may be present in the parent class, it might actually be defined in a descendant class. Note also that the corresponding set/get functions might be in a descendant class when the action depends on the descendant class.

---

## Context Attributes and LI Layers

The Context attributes that the pipeline needs to check at rendering time vary depending on the pipeline layer. A pipeline written at the LI-1 layer needs to implement the complete set of XGL attributes, or at least account for them. At the LI-2 layer, the device pipeline is using the software pipeline to handle some of the processing; therefore, the device pipeline has a smaller subset of attributes that it is accountable for. At the LI-3 layer, the number of attributes that a device pipeline must handle is even smaller. For example, an LI-1 port must handle back surface attributes and transforms, but at the LI-2 level these attributes have been processed by the software pipeline, and the device pipeline no longer needs to concern itself with them. This concept is illustrated in Figure 6-2.

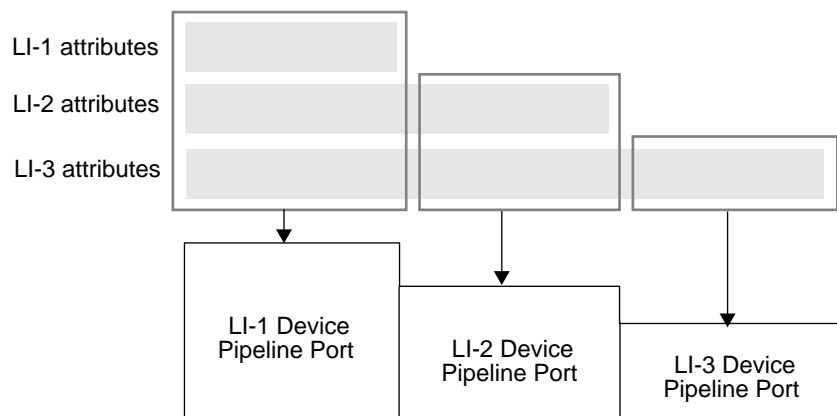


Figure 6-2 Device Pipeline and Layered Attributes

See Chapter 9, “Writing Loadable Interfaces” for information on the attributes used by each LI function.

Note that you will probably want to make use of the `objectSet()` function to optimize Context state retrieval. The `objectSet()` function notifies the device pipeline about changes to Context attributes. If a change occurred, the pipeline must get the new value of the attribute and reload the state into the hardware. In addition, the pipeline uses the stroke tables to get the values of attributes for primitives multiplexed on the multipolyline primitive. See Chapter 5, “Handling Changes to Object State” for information on the `objectSet()` function and stroke groups.

## Getting Attribute Values from the Context

From the `XglDpCtx` object, you can get Context attribute values and values for objects associated with the Context. The `XglDpCtx` object is provided with a pointer to the Context object. This pointer is named `ctx` and is an `XglDpCtx` protected member data. Note that `ctx` already points to a Context of the right dimension. In other words, in `XglDpCtx2d`, `ctx` is already of type `XglContext2d*`, and in `XglDpCtx3d`, `ctx` is already of type `XglContext3d*`, so you don't have to cast the pointer to the correct type. Using the Context pointer, you can get an attribute using `ctx->getAttribute()`.

Example code for a pipeline getting depth cue attributes from a 3D Context might be:

```
Xgl_depth_cue_mode    dc_mode = ctx->getDepthCueMode();

if (dc_mode != XGL_DEPTH_CUE_OFF) {
    float    scale_front;    // Scale factors to use
    float    scale_back;

    if (dc_mode == XGL_DEPTH_CUE_SCALED) {
        float    scale_factors[2]; // XGL DC scale factors
        ctx->getDepthCueScaleFactors(scale_factors);
        scale_front = scale_factors[0];
        scale_back = scale_factors[1];
    }
    else {    // continue
```

## Getting Attribute Values from Other Objects

To render line patterns, markers, and other application-definable data, the device pipeline needs to get information from the objects that the application has associated with the Context. In most cases, handles for these objects are retrieved from the Context object using `ctx->getObject()`. In the following cases, however, the pipeline does not retrieve the object handle for an object from the Context, even though these objects are associated with the Context at the API-level:

- The object handle for the Transform object is retrieved from the view group interface object. See “Getting Information from a Transform Object” on page 125.

- The pipeline is provided with pointers to the Device object in several places. See “Getting Information From the Device Object” on page 145.
- From within `lil/2MultiPolyline()`, the line pattern handle is retrieved from the stroke group. For more information, see “Getting Attribute Values From the Stroke Group Object” on page 125.

Table 6-1 shows the objects that the application can associate with the Context and the `get...()` functions used to retrieve data from them.

*Table 6-1* Getting Information from Xgl Objects

Object	Function
Data Map Texture object (3D only)	<code>getDmapTexture()</code>
Device object	See page 145.
Light object (3D only)	<code>getLight()</code>
Line Pattern object	<code>getLinePattern()</code>
Marker object	<code>getMarker()</code>
Stipple pattern Memory Raster object	<code>getRasterFpat()</code>
Stroke Font object	<code>getSfont()</code>
Texture Map object (3D only)	<code>getTmap()</code>
Transform object	See page 125.

Using the object handle, the pipeline can retrieve attribute data through the public interfaces of the DI object classes.

The following example shows a pipeline accessing a Marker via the Context, using a Marker interface, and getting a Marker attribute from the Context.

```
const XglMarker*    marker;
const XglPrimData* mdata;
float              scale;

marker = ctx->getMarker();
mdata = marker->getActualDescription();
scale = ctx->getMarkerScaleFactor();
```

## Getting Information from a Transform Object

To access member functions of the Transform object, the pipeline gets a handle to the Transform object through the view group interface object. The pipeline is provided with a view group interface object and a pointer to the object named `viewGrpItf` in the `XglPipeCtx{2,3}d` parent class. The pointer to the view group interface object is of type `XglViewGrp2dItf*` or `XglViewGrp3dItf*`, depending on the Context.

To access the Transform, use the pointer to the view group interface object and then access the Transform's interfaces using the handle to the Transform. The following example shows a pipeline using the Transform interface `getMatrixFloat()` from a Transform associated with a 2D Context.

```
XglTransform*          xform;
const Xgli_matrix_f3x3* matrix;

// Load the MC-to-DC transform matrix
xform = (XglTransform*) viewGrpItf->getMcToDc();
matrix = (const Xgli_matrix_f3x3*) xform->getMatrixFloat();
```

See “Transform Interfaces and Flags” on page 137 in this chapter for information on Transform interfaces, and see Chapter 7, “View Model Derived Data” for information on the view group interface object. Note that if the pipeline is not using the derived data facility, it can get Transforms from the Context; see page 150 for more information.

## Getting Attribute Values From the Stroke Group Object

For primitives that are multiplexed on the multipolyline primitive, the XGL core provides a generic group, the stroke group, that holds the necessary attribute information. The stroke group is the source from which the pipeline obtains the values for the line attributes, such as line color, during an `lil/2MultiPolyline()` call. The stroke group attributes that map to API attributes are:

- Antialiasing blend equation
- Antialiasing filter width
- Antialiasing filter shape
- Alternate color

- Cap
- Color
- Color selector
- Join
- Miter limit
- Pattern
- Style
- Width scale factor

The Context object provides a current stroke pointer to indicate which stroke group will be used for rendering. The current stroke pointer points to one of the stroke group objects. When the device pipeline receives a request to render a multipolyline, it gets the pointer to the current stroke group using the Context interface `getCurrentStroke()`:

```
cur_stroke = ctx->getCurrentStroke()
```

The pipeline can then get the attribute values for the attributes from the current stroke group. For example, to get the current value for color, the pipeline calls the stroke group's `getColor()` interface:

```
cur_stroke->getColor()
```

From within curves (for example, `lilMultiArc()`), the pipeline can use `ctx->getLinePattern()` or `ctx->getCurrentStroke()->getPattern()`. See Chapter 5, “Handling Changes to Object State” for more information on getting attribute information through the stroke group.



## Non-API Interfaces Provided in API Objects

The API attributes are documented in the *XGL Reference Manual*; therefore, the interfaces the pipeline can use to retrieve API attribute values are not documented here. However, the device-independent classes provide internal methods to support the pipeline, and these methods are briefly described in this chapter.

### Context Interfaces



See `Context.h` for the `get...()` interfaces you can use to retrieve state values from the Context. The `XglContext` class provides the following internal interfaces.

```
const Xgli_surf_face_attr*
    const getSurfFrontFaceAttr() const
const Xgli_surf_attr_2d* const getSurfAttr() const
```

Functions that enable the pipeline to get general surface attributes within a single structure. These functions can facilitate device pipeline manipulation of surface attributes. See `Context.h` for the structure definitions.

```
Xgl_render_mode getRealRenderBuffer() const
```

This function takes into account the number of buffers allocated (in the case of the Window Raster) and if the Z-buffer is enabled, determines which buffers the pipeline should render into.

```
Xgl_usgn32 getRealPlaneMask() const
```

The real plane mask is the `XGL_CTX_PLANE_MASK` diminished by the bits, which should not be touched in relation to the X color map.

```
Xgl_usgn32 getNewFramePlaneMask()
```

Since the real plane mask prevents regular rendering from changing the bits that XGL does not own in the X pixels, new frame must prepare those bits (in other words, write them once per frame).

```
void
addPickToBuffer(Xgl_usgn32 pick_id1, Xgl_usgn32 pick_id2)
```

Adds a pick event to the device-independent pick buffer.

```
Xgl_boolean checkLastPick() const
```

Compares the last recorded pick IDs with the current pick IDs. Returns TRUE if identical.

```
Xgl_attribute* getAttrTypeListAll() const
```

Returns a list of all 2D and 3D Context attributes.

```
virtual void receive(XglObject* obj, const XglMsg& msg)
```

Used by XGL core only.

## Context 2D Interfaces



See Context2d.h for the `get...()` interfaces you can use to retrieve state values from the Context2d class. The XglContext2D class includes the following internal interfaces:

```
const XglStrokeGroup* getCurrentStroke() const
```

Returns a pointer to the current stroke group.

```
XglDpCtx2d* getDp() {return dp;}
```

Used by XGL core only.

```
XglSwpCtx2d* getSwp() const
```

Used by XGL core only.

```
void assignCurStrokeAsLine()
void assignCurStrokeAsText()
void assignCurStrokeAsEdge()
void assignCurStrokeAsMarker()
void assignCurStrokeAsSurfFront()
```

Sets the Context current stroke pointer to the requested stroke group. For example, `assignCurStrokeAsLine()` causes the Context current stroke pointer to point to the line stroke group.

```
XglViewGrp2dItf* getViewGrp() const
```

Used by XGL core only. The pipeline should not use this function but should use instead the pointer to its own view group interface object in its XglDpCtx object.

## Context 3D Interfaces



See `Context3d.h` for the `get...()` interfaces you can use to retrieve state values from the 3D Context. The `XglContext3d` class includes the following internal interfaces:

```
const Xgli_surf_face_attr_3d* const
    getSurfFrontFaceAttr3d() const
const Xgli_surf_face_attr* const
    getSurfBackFaceAttr() const
const Xgli_surf_face_attr_3d* const
    getSurfBackFaceAttr3d() const
const Xgli_surf_attr_3d* const getSurfFrontAttr3d() const
const Xgli_surf_attr_3d* const getSurfBackAttr3d() const
```

Functions that allow the pipeline to get a number of 3D surface attributes within a single structure. These functions can facilitate device pipeline manipulation of 3D surface attributes. At LI-2, face determination has already taken place. A pipeline can set up the surface attribute pointer based on the facing in the renderer and do all the attribute processing without referring to the actual facing. See `Context3d.h` for the structure definitions.

```
const XglStrokeGroup3d*    getCurrentStroke() const
```

Returns a pointer to the current stroke group.

```
void assignCurStrokeAsLine()
void assignCurStrokeAsText()
void assignCurStrokeAsEdge()
void assignCurStrokeAsMarker()
void assignCurStrokeAsSurfFront()
void assignCurStrokeAsSurfBack()
```

Points the Context current stroke pointer to the requested stroke group. For example, `assignCurStrokeAsLine()` causes the Context current stroke pointer to point to the line stroke group.

```
Xgl_boolean getFrontTexturing() const
```

Returns an *Xgl\_boolean* value, which is `TRUE` if the color type is RGB, if front fill style is other than hollow or empty, and there is at least one active front Data Map Texture object in the Context.

```
Xgl_boolean getBackTexturing() const
```

Returns an *Xgl\_boolean* value, which is `TRUE` if the color type is RGB, if back fill style is other than hollow or empty, and there is at least one active back Data Map Texture object in the Context.

```
Xgl_boolean getTlistEdgeFlag() const
```

If NURBS edge flags are on and the device pipeline calls the software pipeline to render a NURBS surface, the software pipeline calls the device pipeline `lilQuadrilateralMesh()` or `lilTriangleStrip()`. The software pipeline uses `ctx->setTlistEdgeFlag()` to inform the pipeline primitives whether they should show the edges of tessellated triangle lists. The functions `lilQuadrilateralMesh()` or `lilTriangleStrip()` can access `tlistEdgeFlag` by calling `ctx->getTlistEdgeFlag()`. The default value is `FALSE`.

```
virtual void receive(XglObject* obj, const XglMsg& msg)
```

Used by XGL core only.

## Data Map Texture Interfaces



See `DmapTexture.h` for the `get...()` interfaces you can use to retrieve state values from the Data Map Texture object. The `XglDmapTexture` class includes the following internal functions:

```
Xgl_texture_desc* const* getDescriptors() const
```

Returns a pointer to the texture descriptors (that are read-only) in a Data Map Texture object. This is similar to the function `getDescriptors(Xgl_texture_desc[])`, except that in this case the device pipeline has to allocate space for the texture descriptors and a copy of the texture descriptors is returned as opposed to a pointer.

```
virtual void receive(XglObject* obj, const XglMsg& msg)
```

Used by XGL core only.

## Device Interfaces



See `Device.h` for the `get...()` interfaces you can use to retrieve state values from the Device object. The `XglDevice` class includes the following internal functions.

```
Xgl_vdc_orientation getDcOrientation() const
```

Used by XGL core only.

```
XglDpDev* getDpDev() const
```

Returns a pointer to the `XglDpDev` object.

```
XglDrawable* getDrawable() const
```

Returns the drawable associated with the Device.

```
float getGammaValue() const
```

Returns the gamma value of the device needed by the software pipeline to implement gamma correction for antialiased stroke primitives.

```
float* getGammaPowerTable() const
```

Returns a pointer to the `gammaPowerTable`. The *i*th entry of the table is the value  $i/255.0$  raised to the power of the gamma value. The size of the table is 256 entries.

```
float* getGammaInversePowerTable() const
```

Returns a pointer to the `gammaInversePowerTable`. The *i*th entry of the table is the value  $i/255.0$  raised to the power of the reciprocal of the gamma value. The size of the table is 256 entries.

## Light Interfaces



See `Light.h` for the `get...()` interfaces you can use to retrieve state values from the `Light` object. The `XglLight` class includes the following internal functions:

```
const Xgl_pt_f3d& getNegDirection() const
```

For lights of type `DIRECTIONAL` (`XGL_LIGHT_DIRECTIONAL`) and `SPOT` (`XGL_LIGHT_SPOT`), this function returns the vector opposite to the direction of propagation for directional lights, or opposite to the light ray on the central axis for spot lights, in other words, the negative of `XGL_LIGHT_DIRECTIONAL`, or pointing toward the light source.

```
float getCosAngle2() const
```

For lights of type `SPOT` (`XGL_LIGHT_SPOT`), this function returns the cosine of half the spot angle, in other words, the angle between the central axis of the spot light and any ray at the boundary of the cone of illumination.

## Line Pattern Interfaces



See `LinePattern.h` for the `get...()` interfaces you can use to retrieve state values from the `Line Pattern` object. The `XglLinePattern` class includes the following internal functions.

---

**Note** – In most cases, you will get to `Line Pattern` data via the stroke group. For example, use `ctx->getCurrentStroke()->getPattern()`.

---

```
void getActualData (float*) const
```

Copies the actual line pattern data. The actual data differs from the API data in that it is always `float`, and it includes the odd-length processing.

```
const float* getActualData() const
```

Returns a pointer to the actual line pattern data, including the odd-length processing.

```
Xgl_usgn32 getActualDataSize() const
```

Returns the size of the actual line pattern data.

```
float getActualOffset() const
```

Returns the offset in the actual line pattern data.

```
float getLength() const
```

Returns the total length of the line pattern in actual data.

```
Xgl_usgn32 getStartSeg() const
```

Returns the segment in actual data where the offset is.

```
float getStartSegRemain() const
```

Returns the remaining length in the segment in actual data at the offset location.

## Marker Interfaces



See `Marker.h` for the `get...()` interfaces you can use to retrieve state values from the Marker object. The `XglMarker` class includes the following internal function:

```
const XglPrimData* getActualDescription() const
```

Returns a pointer to the `XglPrimData` description of the marker.

## MipMap Texture Interfaces



See `MipMapTexture.h` for the `get...()` interfaces you can use to retrieve state values from the MipMap Texture object. The `XglMipMapTexture` class includes the following internal function:

```
Xgl_usgn8 getElement(Xgl_usgn32 level, Xgl_usgn32  
                    channel_num, Xgl_usgn32 x, Xgl_usgn32 y)
```

Returns the contents of the channel *channel\_num* at position (x,y) from the level *level* in the MipMap.

## Raster Interfaces



See `Raster.h` for the `get...()` interfaces you can use to retrieve state values from the Raster object. The `XglRaster` class includes the following internal interfaces:

```
void setDoPixelMapping (Xgl_boolean b)
```

Used by the Memory Raster device pipeline only. Differentiates between a “real” Memory Raster device (`b` is `FALSE`) and a backing store Memory Raster (`b` is `TRUE`).

```
Xgl_boolean getDoPixelMapping() const
```

Used by `RefDpCtx`, a Memory Raster, and the software pipeline to determine if `DoPixelMapping` has been set.

## Texture Map Interfaces



See `Tmap.h` for the `get...()` interfaces you can use to retrieve state values from the Texture Map object. The `XglTmap` class includes the following internal functions:

```
Xgl_texture_general_desc* const* getDescriptors() const
```

Returns a pointer to the texture descriptors (that are read-only) in a Texture Map object. This is similar to the function `getDescriptors(Xgl_texture_general_desc[])`, except that in this case the device pipeline has to allocate space for the texture descriptors and a copy of the texture descriptors is returned as opposed to a pointer.

```
virtual void receive(XglObject* obj, const XglMsg& msg)
```

Used by XGL core only.



## Window Raster Interfaces



See `RasterWin.h` for the `get...()` interfaces you can use to retrieve state values from the Window Raster object. The `XglRasterWin` class includes the following internal functions:

```
void setDgaCmapPutFunc(void(*PutFunc)(Dga_cmap dga_cmap,
                                     int inden, int count, u_char* red,
                                     u_char* green, u_char* blue)
```

Provided by the XGL core so that a device pipeline can register a callback function to update the hardware color map. For more information on `PutFunc`, see the documentation for `dga_cm_write()` in the *OpenWindows Server Device Developer's Guide*.

```
XglPixRectMem* getSwZBuffer() const
```

Returns a pointer to the `XglPixRectMem` object that represents the software Z-buffer.

```
XglPixRectMem* getSwAccumBuffer() const
```

Returns a pointer to the `XglPixRectMem` object that represents the software accumulation buffer.

```
virtual void receive(XglObject* obj, const XglMsg& msg)
```

Used by XGL core only.

## Memory Raster Interfaces



See `RasterMem.h` for the `get...()` interfaces you can use to retrieve state values from the Memory Raster object. The `XglRasterMem` class provides the following internal functions:

```
XglPixRectMem* getImageBufferPixRect() const
```

Returns a pointer to the `XglPixRectMem` object that represents the image buffer for the memory raster.

```
XglPixRectMem* getZBufferPixRect() const
```

Returns a pointer to the `XglPixRectMem` object that represents the Z-buffer for the memory raster.

```
XglPixRectMem* getAccumBufferPixRect() const
```

Returns a pointer to the `XglPixRectMem` object that represents the accumulation buffer for the memory raster.

```
Xgl_usgn32 getImgBufLineBytes() const
```

Gets the value for `linebytes` for the image buffer when the memory raster is set up to access memory for retained windows. `linebytes` is the number of bytes that separates one line in a raster, in other words, the number of bytes from `(x,y)` to `(x,y+1)`.

```
void syncRtnDevice(XglRasterWin*)
```

Used by the XGL core only.

```
virtual void receive(XglObject* obj, const XglMsg& msg)
```

Used by XGL core only.

## Stroke Font Interfaces



See `Sfont.h` for the `get...()` interfaces you can use to retrieve state values from the Stroke Font object. The `XglSfont` class includes the following internal functions:

```
Xgl_boolean getIsFontLoaded() const
```

Returns a Boolean value that indicates whether the font file is actually loaded.

```
Xgl_sfont_data* getSfontData()
```

Loads the font file and returns a pointer to the data.

```
Sfont_inst* getSfontInst()
```

Returns a pointer to the actual strokes that define an entire font.

## Transform Interfaces and Flags

### Transform Flag

The Transform object maintains a member datum called *flag* that contains internal information for the XglTransform class. The pipeline can get the flag information by calling the `getFlag()` function. The flag consists of the values in the enumerated type *Xgli\_trans\_flag*. Most of the flag bits are used to keep track of the state of the Transform, but two of the bits, `XGLI_TRANS_SINGULAR` and `XGLI_TRANS_INVERSE_VALID`, may be of use to the device pipeline.

If the `XGLI_TRANS_SINGULAR` bit is set, this indicates that the matrix is singular and that the application, the XGL core, or the device pipeline has attempted to invert it. However, if the bit is not set, this does not necessarily mean that the matrix is nonsingular but may simply mean there has not been an attempt to take the inverse of the matrix. The `XGLI_TRANS_INVERSE_VALID` bit works similarly. Table 6-2 shows the relationship between these two bits and what the bit settings mean.

Table 6-2 XGLI\_TRANS\_SINGULAR

SINGULAR	INVERSE VALID	Meaning
0	0	The inverse of the matrix has not been taken, so information on its singularity is not available.
1	0	Singular matrix.
0	1	Nonsingular matrix.
1	1	Not possible.

### Transform Member Records

The Transform member datum *memberRecord* defines the matrix groups to which a matrix is a member. If the application has specified the membership of a matrix to a matrix group with `xgl_transform_write_specific()` or the application constructs its Transforms with XGL's transform utilities, such as `xgl_transform_scale()`, the device pipeline can use the

`getMemberRecord()` function to determine which groups the matrix belongs to and take advantage of that information to speed up the processing of transformations.

The member datum *memberRecord* holds combinations of the macros `XGL_TRANS_GROUP_xx` defined in `xgl.h`. The groups are:

```
/* Transform groups */
typedef      Xgl_usgn32      Xgl_trans_group;
#define XGL_TRANS_GROUP_IDENTITY      0x001
#define XGL_TRANS_GROUP_TRANSLATION  0x002
#define XGL_TRANS_GROUP_SCALE        0x004
#define XGL_TRANS_GROUP_ROTATION     0x008
#define XGL_TRANS_GROUP_WINDOW       0x010
#define XGL_TRANS_GROUP_SHEAR_SCALE  0x020
#define XGL_TRANS_GROUP_LENGTH_PRESERV 0x040
#define XGL_TRANS_GROUP_ANGLE_PRESERV 0x080
#define XGL_TRANS_GROUP_AFFINE       0x100
#define XGL_TRANS_GROUP_LIM_PERSPECTIVE 0x200
```

Each group has special properties. XGL takes advantage of these for more efficient operations, including inversion and multiplication of two matrices, multiplication of a matrix by points, and multiplication of a matrix by normal vectors. The device pipeline can use the groups to classify a matrix in order to apply optimized operations during rendering.

Applications can specify the member record using the Transform operator `xgl_transform_write_specific()` to write a matrix into a Transform. This operator takes a parameter of type *Xgl\_trans\_member*, which is defined in `xgl.h` and which specifies the groups the matrix belongs to. In addition, XGL maintains the member record when applications use XGL's utilities for constructing Transforms. If applications use their own utilities for constructing matrices, and they do not specify the member record when they write a matrix into a Transform, the member record is not maintained because analyzing the matrix is time-consuming. For information on the member groups, see the *XGL Reference Manual* page on `xgl_transform_write_specific()`.

The device pipeline can test the member flags by calling the `getMemberRecord()` function of the Transform object, as in the following pseudocode example. Note that in this example, the application has not supplied the `w` value.

```
Xgl_trans_group      group = transform->getMemberRecord();
const Xgl_matrix_f3d*mat = (const Xgl_matrix_f3d*)
                        transform->getMatrixFloat();

if (group & XGL_TRANS_GROUP_IDENTITY) {
// identity
    x1 = x ; y1 = y ; z1 = z ; w1 = 1 ;
}
else if (group & XGL_TRANS_GROUP_TRANSLATION) {
// translation
    x1 = x + (*mat)[3][0] ;
    y1 = y + (*mat)[3][2] ;
    z1 = z + (*mat)[3][2]; w1 = 1 ;
}
}
```

It is the device pipeline's responsibility to check the membership record from the Transform object. Derived data calculates the correct matrix, but a device pipeline may or may not be able to use that matrix in certain circumstances. The device pipeline can take advantage of knowing that certain flag bits are set.

### ***Example: Checking a Member Record for Identity***

One example of using a member record is to determine whether a transform is identity, as shown in this code sample:

```
const XglTransform* mc_to_cc = viewGrpItf->getMcToCc();

if (mc_to_cc->getMemberRecord() & XGL_TRANS_GROUP_IDENTITY) {
    // transform is identity
}
else {
    // transform is not identity
}
```

### ***Example: Checking a Member Record to Do Lighting***

A device may implement lighting in Model Coordinates (MC) for improved performance over World Coordinates (WC) and Lighting Coordinates (LC) because the normal vectors do not need to be transformed to do lighting in MC (see Chapter 7, “View Model Derived Data” for a description of coordinate systems). Lighting can be performed correctly in MC only if the Model Transform preserves angles. The reason is that lighting is specified in WC in the conceptual view model so lighting must be performed in a coordinate system related to WC by an angle-preserving matrix; otherwise, the dot products in lighting calculations do not equal those in WC. You must check the member record of the Model Transform as discussed below before you attempt to perform lighting calculations in Model Coordinates.

The membership record of the Model Transform can give additional information relevant to lighting calculations. The Model Transform may have the following properties:

1. Length-preserving
2. Angle-preserving
3. Affine with anisotropic scaling
4. Perspective
5. Singular

The order of these properties for lighting is from easiest to hardest, which is the sequence that you should apply to testing the Model Transform. The testing needs to be performed at least once whenever the Model Transform changes. The view group interface can assist detection of these changes; see Chapter 7, “View Model Derived Data” for information on the view group interface.

When the Model Transform preserves lengths (and angles), you can perform lighting in MC. However, if the device always performs lighting in WC or LC, then the device would transform normal vectors as column vectors by the 3×3 upper-left submatrix of the WC-to-MC or LC-to-MC Transform respectively. Lighting calculations generally require unit length vectors, so XGL requires applications to supply unit length normal vectors in MC. If the Model Transform preserves lengths, then the normal vectors in WC and LC have unit length, so you do not have to readjust their lengths after transformation. To

determine whether the Model Transform preserves lengths, you can check the membership record to see if the `XGL_TRANS_GROUP_LENGTH_PRESERV` bit is set.

When the Model Transform preserves angles, you can perform lighting in MC. However, if the device always performs lighting in WC or LC, you can transform normal vectors the same way as for the length-preserving case. Unit normal vectors in MC transformed in this manner by an angle-preserving matrix always have the same length after transformation. This length is the isotropic scale factor, which can be obtained with `getIsotropicScale()`. If you need a unit vector after transformation by an angle-preserving matrix, just divide the transformed vector by the isotropic scale factor. If you have several vectors that you want to transform by an angle-preserving matrix and you need unit length vectors after the transformation, then you can divide all the elements of the matrix by the isotropic scale factor, and the unit length vectors transformed by this matrix will yield unit length vectors. This saves you from dividing all the vectors by the isotropic scale factor. To determine whether the Model Transform preserves angles, you can check the membership record to see if the `XGL_TRANS_GROUP_ANGLE_PRESERV` bit is set.

When the Model Transform is affine and does not preserve angles, you must perform lighting in WC or LC. In this case, the Model Transform scales geometry anisotropically, and the amount of scaling depends on the direction. After transforming a unit vector by an affine matrix with anisotropic scaling, you need to calculate the length of the transformed vector and divide the vector by its length. Calculation of the length usually requires a square root, but a lookup table may be faster and accurate enough. To determine whether the Model Transform is affine and does not preserve angles, you can check the membership record to see if the `XGL_TRANS_GROUP_AFFINE` bit is set after checking that the `XGL_TRANS_GROUP_ANGLE_PRESERV` bit is not set.

When a Model Transform has perspective, lighting calculations are correct only in WC or LC. The calculation of transformed normal vectors is difficult, and most applications do not use Model Transforms with perspective. You can assume that this situation never occurs. If it does occur, treat it as if the Model Transform is affine with anisotropic scaling. XGL has the limitation that the Model Transform cannot have perspective.

When a Model Transform is singular, lighting calculations cannot be performed except for ambient lighting. In MC, the light positions and eye position or vector cannot be calculated. In WC or LC, the normal vectors cannot be

calculated uniquely. Therefore, you should only do ambient lighting. To determine whether the Model Transform is singular, get its flag as described on page 137.

## Transform Internal Interfaces



See `Transform.h` for the `get...()` interfaces you can use to retrieve state values from the Transform object. The `XglTransform` class includes the following internal functions.

---

**Note** – The pipeline can access transforms and view group attributes via the Context object if the pipeline never falls back on the software pipeline. If the device pipeline never uses the software pipeline, it can access the Local Model Transform using `XglTransform* mt = ctx->getModelTrans()`. Even in this case, however, it is a good idea for pipelines to use derived data so that they work consistently with other pipelines. So the preferred way to access the Local Model Transform is to use derived data, as in `mt = (XglTransform*) viewGrpItf->getMcToWc()`. Then, to get to the interfaces for the Transform, use `mt->getxx`.

---

```
Xgli_trans_flags getFlag()
```

Returns the transform flag described on page 137.

```
Xgl_trans_group getMemberRecord()
```

Returns a word containing the matrix groups. See page 137 for information on the internal flag *memberRecord* and see the `xgl_transform_write_specific()` man page for information on the member groups.

```
const float* getMatrixFloat()
```

Returns a floating-point representation of a single-precision matrix. Note that when an application uses `xgl_transform_write_specific()` to write a matrix, it passes a pointer to a matrix type defined in `xgl.h`. However, the internal matrix types defined in `Transform.h` are the actual representations of the matrix data that the application will get. Although the API documentation states that 2D matrices are 3×2 and in `xgl.h` the matrix



type *Xgl\_matrix\_f2d* is defined as a float 3×2 matrix, internally the 2D matrices are 3×3 matrices of type *Xgli\_matrix\_i3x3*, *Xgli\_matrix\_f3x3*, or *Xgli\_matrix\_d3x3*.

Note that XGL does calculations in double-precision format and only converts to single-precision format when the application or pipeline requests it by calling `getMatrixFloat()`.

```
const float* getMatrix()
```

Equivalent to `getMatrixFloat()`; however, `getMatrixFloat()` is the recommended version.

```
const double* getMatrixDouble()
```

Returns a double-precision matrix.

```
const Xgl_sgn32* getMatrixInt()
```

Returns a 32-bit integer version of the matrix.

```
double getIsotropicScale()
```

Returns a value that indicates how much scaling the matrix does when it scales isotropically. This is valid only when the `XGL_TRANS_GROUP_ANGLE_PRESERV` bit is set in the member record.

```
double getNorm()
```

Returns the mathematical norm of a matrix.

```
double getNormInverse()
```

Returns the reciprocal of the norm of a matrix.

```
copyConvert()
```

Used internally by `getMatrixInt()`.

```
void transPt(const Xgli_pt*, Xgli_pt*)
```

Transforms a single point of type *Xgli\_pt* (defined in `Transform.h`) and stores it in a different block of memory. This is different than the API `xgl_transform_point()` function, which transforms a point and overwrites the original block of memory.

```
void transPtList(const Xgli_pt_list*, Xgli_pt_list*)
```

Transforms a point list of type *Xgli\_pt\_list* (defined in `Transform.h`) and stores it in a different block of memory. This is different than the API `xgl_transform_point_list()` function.

```
void transNormal(const Xgl_pt_f3d* src, xgl_pt_f3d* dest)
```

Transforms a normal of type *Xgl\_pt\_f3d* and returns a normal of the same type.

```
void
```

```
transUnitNormal(const Xgl_pt_f3d* src, Xgl_pt_f3d* dest)
```

Transforms a unit normal of type *Xgl\_pt\_f3d* and returns a unit normal of the same type.

```
void
```

```
transUnitNormalDouble(const Xgl_pt_d3d* src,  
                      Xgl_pt_d3d* dest)
```

Transforms a unit normal of type *Xgl\_pt\_d3d* and returns a unit normal of the same type.

## Getting Information From the Device Object

The device pipeline may need to get information from the device-independent Device object, from the Drawable associated with the Device, or from a Color Map object that the application has associated with the Device. Pointers to the Device object are available as follows:

- The `XglDpDev` object has a member data `device` that holds a pointer to the device-independent Device object. Note that `device` already points to a Device of the right type. In other words, in an `XglDpDevWinRas` object, `device` is already `XglRasterWin*`, in an `XglDpDevMemRas` object, `device` is already `XglRasterMem*`, and in an `XglDpDevStream` object, `device` is already `XglStream*`.
- The `XglDpCtx` object has a member data `device` that holds a pointer to the device-independent Device object.

A pointer to the Drawable object is available in the `XglDpCtx` object. Your pipeline `XglDpCtx` can use the `drawable` pointer rather than using `getDevice()->getDrawable()`. Note that the `device` and `drawable` pointers are fixed at `XglDpDev` creation and do not change for the life of the `XglDpDev` object.

To get a handle to the Color Map object, use the inline function `getCmap()`, as in `device->getCmap()`. The `getCmap()` function is defined in `Device.h`.

If you frequently use some object pointers (or even data itself), you can cache the object pointers, provided that you use the `objectSet()` function correctly to stay synchronized with Context state changes. A good example of this is caching a pointer to the color map object. If you think frequent use of `device->getCmap()` is too time consuming, save the return value in a member data of your own `XglDpCtx`.

## Color Map Interfaces

Because the Color Map object is set on the Device, its interfaces are available through the Device object. With a handle to the Color Map object, the pipeline can access the Color Map's interfaces as follows:

```
device->getCmap()->getColorMapper()
```

Note that the application can change the color map by setting a new Color Map object on the Device or by changing an existing color map. The pipeline is notified of color map changes by the message passing mechanism (see page 99) and by a direct `dpDev->setCmap(cmap)` call from the Context to the pipeline `XglDpDev` object (see page 54). Although the color map can change during program execution, applications are not allowed to change the device's color type after the device has been created.



See `Cmap.h` for the `get...()` interfaces you can use to retrieve state values from the Color Map object. The `XglCmap` class includes the following internal functions.

```
Xgl_usgn32 getPlaneMaskMask() const
```

Returns an *Xgl\_usgn32* value in which the bits set indicate the knowledge XGL has of the bits where rendering is allowed, with regard to the XGL and X colormaps.

```
Xgl_color* getColorTable() const
```

This function returns a pointer to the first color of the color table.

```
XglCmapDrawable* getCmapDrawable() const
```

Used by XGL core only.

```
Xgl_usgn32 lookUpDitherValue(Xgl_usgn32, Xgl_usgn32)
```

Returns the dither matrix value at the given position.

```
Xgl_sgn32 lookUpInternalDitherValue(Xgl_usgn32, Xgl_usgn32)
```

Returns the value of the internal dither matrix at the given position. The internal dither matrix is the transpose of the regular dither matrix in “fract24” (s7.24) format, and it is divided by 255 (so the renderer need not divide). The internal dither matrix is valid only when the dither matrix is 8×8. It is the transpose of the regular matrix to allow scanline access.

```
Xgl_sgn32*
```

```
lookUpInternalDitherAddress(Xgl_usgn32, Xgl_usgn32)
```

Returns the address of the value of the internal dither matrix at the given position.

---

```
Xgl_boolean getMapperHasBeenSet() const
```

Indicates whether color mapper has been set. Used by the XGL core only.

```
Xgl_boolean getInverseMapperHasBeenSet() const
```

Indicates whether the inverse color mapper has been set. Used by the XGL core only.



## *View Model Derived Data*

---

7 

This chapter describes how a device pipeline gets data for implementing the view model. The chapter includes information on the following topics:

- Overview of view model derived data
- A summary of how derived data is implemented
- Information on how pipelines access derived data information



As you read this chapter, you will find it helpful to have access to the header files for the derived data mechanism. These are:

- `ViewCache.h`, `ViewCache2d.h`, and `ViewCache3d.h`
- `ViewConcern2d.h` and `ViewConcern3d.h`
- `ViewGrp2d.h` and `ViewGrp3d.h`
- `ViewGrp2dItf.h` and `ViewGrp3dItf.h`
- `ViewGrp2dConfig.h` and `ViewGrp3dConfig.h`

## Overview of View Model Derived Data

XGL defines a conceptual view model consisting of a number of coordinate systems in which an application can specify certain operations. These coordinate systems are Model Coordinates (MC), World Coordinates (WC), Virtual Device Coordinates (VDC), and Device Coordinates (DC). Examples of the usage of coordinate systems in the 3D view model include specification of geometry in MC, lights and model clip planes in WC, view clip planes and depth cue reference planes in VDC, and the pick aperture in DC. The coordinate systems are related by a sequence of transformations. The Local Model and Global Model Transforms are concatenated to form the Model Transform, which maps geometry from MC to WC. The View Transform maps geometry from WC to VDC. The VDC map, VDC orientation, VDC window, DC orientation, DC viewport, and jitter offset collectively define a mapping between VDC and DC. This view model is conceptual because an application can think of an operation as occurring in the coordinate system where it is specified, but a pipeline actually may implement the operation in another coordinate system for improved performance as long as the results are equivalent.

XGL provides a facility to assist pipelines with implementation of the view model's operations. The facility is named *view model derived data* or simply *derived data*. Derived data maintains a cache of items derived from a Context's view model attributes. The derived items include Transforms for mapping geometry between coordinate systems as well as items in various coordinate systems such as the view clip bounds, lights, eye positions or eye vectors, model clip planes, and depth cue reference planes. For example, derived data calculates the VDC-to-DC Transform from the Context attributes for the VDC map, VDC orientation, VDC window, DC viewport, and jitter offset, and the Device attribute for DC orientation. In turn, the MC-to-DC Transform is the concatenation of the Model, View, and VDC-to-DC Transforms. This illustrates that a derived item can depend on only API attributes, on only derived items, or on a combination of both.

---

**Note** – Pipelines have the option of not using the derived data facility if and only if the pipeline never falls back on the software pipeline. If the pipeline does fall back on the software pipeline, for example for the processing of annotation text, markers, 2D circles and arcs, or NURBS curves and surfaces, it must use derived data. See “Entry of Geometry from Multiple Coordinate Systems” on page 152 for information.

---



Derived data implements a large collection of items. The items were selected by analyzing the requirements of a theoretical pipeline and by looking at the needs of several graphics devices. The theoretical pipeline employs two coordinate systems that are not exposed at the API level: Lighting Coordinates (LC)<sup>1</sup> and Clipping Coordinates (CC). It also performs operations in several coordinate systems; for example, model clipping operations occur in four coordinate systems so the clip planes are needed in each of these.

Derived data is efficient and easy for device pipelines to use. In particular, derived data is designed for hardware devices that retain the state of the view model such as matrices and clip planes. These device pipelines need to know when a derived item has changed so that the pipeline can reload the item into the device. The calculations are transparent to pipelines, and the design avoids redundancies and extraneous evaluation of derived data items.

The XGL software pipeline also uses derived data. The only difference between many device pipelines and the software pipeline is that the latter does not retain state so it does not need to be informed of changes to derived items. The software pipeline simply gets a derived item as needed, but this does not necessarily cause re-evaluation since the item may be valid already.

## *Design Goals of Derived Data*

The design goals of derived data are:

1. Support geometry entering LI-1 from other coordinate systems (in addition to Model Coordinates) with a simple interface for pipelines.
2. Provide a fast test to inform a pipeline of changes to derived items of concern to that pipeline and minimize data transfer to devices that retain state.
3. Defer calculation of a derived item until a pipeline requests that item, and avoid redundant calculations.

---

1. For information on Lighting Coordinates, see Salim S. Abi-Ezzi and Michael J. Wozny, "Factoring a Homogeneous Transformation for a More Efficient Graphics Pipeline," *Computer Graphics Forum*, North-Holland, Vol. 9, 1990, pp. 245-255.

### *Entry of Geometry from Multiple Coordinate Systems*

The need to support entry of geometry to LI-1 primitives from coordinate systems other than Model Coordinates greatly complicates the design of derived data. As an example, consider the NURBS surface code in the software pipeline. The software pipeline's 3D LI-1 NURBS surface primitive takes control points and knots in MC and produces polylines, triangles, and quadrilateral meshes in LC, CC, and DC. Then this geometry enters the LI-1 primitives of a number of devices. Rather than force device pipeline developers to produce an LI-1 primitive for each coordinate system from which geometry can enter, derived data "fools" pipelines into "thinking" that they are always getting geometry in MC, even when geometry enters from another coordinate system. But this could be expensive if it isn't done carefully.

Consider a simple example of 2D annotation text in the software pipeline. The application passes a character string and a reference point to XGL at the API level. If the device pipeline cannot handle annotation text at LI-1, the software pipeline transforms the reference point from MC to VDC, checks that the point is within the view clip bounds, and constructs a polyline description of the text based on information stored in font files. Derived data provides functions so that a primitive can push the current coordinate system onto a stack and set it to another: VDC in the case of annotation text. Then the primitive can call LI-1 multipolyline. When the multipolyline function requests a transform, for example the MC-to-CC Transform, derived data returns the appropriate transform for the current coordinate system: in this case, the VDC-to-CC Transform. The LI-1 multipolyline primitive doesn't need to be aware that the current coordinate system is VDC instead of MC. When control returns from the LI-1 polyline primitive to the software pipeline's LI-1 annotation text primitive, the latter can pop the coordinate system to restore the original one (which should be MC).

For certain primitives, the software pipeline uses derived data to transform the geometry through part of the pipeline before changing the coordinate system and passing the partially processed geometry on to another LI-1 primitive. An important corollary is that if a device pipeline ever falls back on the software pipeline, the device pipeline must use derived data. If a device pipeline never falls back on the software pipeline, then the device pipeline has the option of using or not using derived data.

## *Changes to Derived Items*

Derived data has a fast test to allow pipelines to determine when at least one derived item has changed since the last time that the pipeline accessed any item. This test especially benefits pipelines whose devices retain view model state. A pipeline can express concern about changes to a specified set of items. This allows pipelines to filter out irrelevant changes, which is important because derived data consists of a large number of items and pipelines typically need only a few items.

A derived item can change as a result of an application changing a view model attribute. A derived item may depend directly or indirectly on that attribute. An attribute change invalidates some previously calculated items.

A change to a derived item can also be the result of a change in the current coordinate system. In the annotation text example in the previous section, derived data returns the VDC-to-CC Transform when the current coordinate system is VDC and the pipeline requests the MC-to-CC Transform. If the previous coordinate system was MC when the pipeline requested the MC-to-CC Transform, the actual MC-to-CC matrix would have been loaded into the device. The change in coordinate system from MC to VDC means that the pipeline needs to load a difference matrix to achieve the MC-to-CC mapping. Hence, a change in the current coordinate system results in changes to derived items even though no items have been invalidated, as in the case of API attribute changes. A pipeline does not need to be aware of the reason for a change in an item. Derived data simply informs the pipeline when an item must be reloaded into a device that retains state.

In addition to the fast test for the whole set of specified items, derived data has a test for each individual item. If the fast test is positive, then at least one of the specified items has changed. The pipeline then needs to check each of the specified items for changes. The pipeline should get a changed item from derived data and reload that state into the device.

The fast test never misses a change to a derived item resulting from a change to an API view model attribute or a change in the current coordinate system. However, the test may be falsely positive because changes to items resulting from changes in the coordinate system cannot be determined quickly with complete accuracy. Hence, the test is overly cautious. Fortunately, the tests for the individual items are quick and completely accurate so derived data eliminates extraneous transfers of state to devices. Since pipelines typically need only a few items, the overhead is not large.

### *Deferred Calculation*

Derived data defers calculations until a pipeline requests a particular item. Each item is a node in an acyclic directed graph of dependencies with API view model attributes at the bottom. When a pipeline requests a particular item, derived data descends the graph until it finds valid items (API view model attributes are always valid) and ascends the graph as it performs calculations until it reaches the requested item. Consequently, if that item is already valid, then no calculation is required. A pipeline's request for a particular item is the trigger for any necessary calculations.

Deferred evaluation has the advantage of eliminating unnecessary calculations. Derived data calculates an item only when a pipeline explicitly requests that item (or one that depends on it) and that item needs to be reloaded into a device because of a change to a relevant API attribute or a change in the current coordinate system. A pipeline is not penalized with expensive calculations if it does not use derived data.

### *Derived Data Items*

Derived data maintains transformations between coordinate systems and maintains a variety of other items that change when Context or Device attributes change.

### *Coordinate Systems and Transforms*

The majority of items in derived data are Transforms for mapping geometry between pairs of coordinate systems. Each Transform has a name of the form "AcToBc" for transforming points from the "A" coordinate system to the "B" coordinate system. For 3D, the "AcToBc" Transform can be used to transform normal and direction vectors from BC to AC by applying these vectors as  $3 \times 1$  column vectors to the  $3 \times 3$  upper-left submatrix of AcToBc's  $4 \times 4$  matrix.

Table 7-1 lists the coordinate systems that derived data supports in 2D.

*Table 7-1 Derived Data 2D Coordinate Systems*

<b>Mnemonic</b>	<b>Name</b>
MC	Model Coordinates
GMC	Global Model Coordinates
WC	World Coordinates
VDC	Virtual Device Coordinates
CC	Clipping Coordinates
DC	Device Coordinates

Table 7-2 lists the coordinate systems that derived data supports in 3D.

*Table 7-2 Derived Data 3D Coordinate Systems*

<b>Mnemonic</b>	<b>Name</b>
MC	Model Coordinates
GMC	Global Model Coordinates
WC	World Coordinates
LC	Lighting Coordinates
IC	Intermediate Coordinates
VDC	Virtual Device Coordinates
CC	Clipping Coordinates
DC	Device Coordinates

**Note** – GMC is the coordinate system after the Local Model Transform and before the Global Model Transform.

In 3D the view-clipping volume in Clipping Coordinates has the boundaries  $[-1,1] \times [-1,1] \times [-1,1]$  with the clip planes at the boundaries. The x-, y-, and z-axes are parallel to those in VDC and DC. The orientation always has the x-axis pointing right, the y-axis pointing up, and the z-axis pointing toward. This is independent of the orientations of VDC (specified by the application) and DC (specified by the device). 2D is similar except that there is no z-axis.

In 3D the View Transform often can be factored into the following form:

$$V = EQG$$

where E is Euclidean (meaning that it preserves the distances between points and the angle between direction vectors), and Q and G are sparse. See the paper by Abi-Ezzi and Wozny for a full description of the decomposition and properties of the coordinate systems. The coordinate system between E and Q is called Lighting Coordinates, and the one between Q and G is called Intermediate Coordinates. The software pipeline NURBS primitives use these coordinate systems, and device pipelines may benefit from them as well.

## Other Derived Items

Derived data maintains a number of items other than Transforms. The view clip bounds in MC, VDC, CC, and DC, and the viewport in DC are available to both 2D and 3D pipelines. 3D pipelines can also access the lights in MC and LC, the eye vector or position in MC, LC, VDC, and CC, a flag indicating when the view projection is parallel as opposed to perspective, the model-clip planes in MC, LC, CC, and DC, the depth cue planes in CC and DC, and a flag indicating when the View Transform can be factored. Table 7-3 lists the items other than Transforms that derived data maintains.

*Table 7-3* Other Items in Derived Data

Mnemonic	Name
VclipBoundsMC	View-clip bounds in MC
VclipBoundsVdc	View-clip bounds in VDC
VclipBoundsCc	View-clip bounds in CC
VclipBoundsDc	View-clip bounds in DC
ViewportDc	Viewport in DC
LightsMc	Lights in MC
LightsLc	Lights in LC
EyeMc	Eye vector or point in MC
EyeLc	Eye vector or point in LC
EyeVdc	Eye vector or point in VDC
EyeCc	Eye vector or point in CC

Table 7-3 Other Items in Derived Data (Continued)

Mnemonic	Name
ParallelProj	Parallel projection flag
MclipPlanesMc	Model-clip planes in MC
MclipPlanesLc	Model-clip planes in LC
MclipPlanesCc	Model-clip planes in CC
MclipPlanesDc	Model-clip planes in DC
DcuePlanesCc	Depth cue planes in CC
DcuePlanesDc	Depth cue planes in DC
ViewCanonical	View canonical flag

## Overview of Derived Data's Implementation

The view model derived data facility consists of a set of four classes for each of 2D and 3D. Table 7-4 lists the class names.

Table 7-4 View Model Derived Data Classes

Generic Name	2D C++ Class Name	3D C++ Class Name
View cache	XglViewCache2d	XglViewCache3d
View group configuration	XglViewGrp2dConfig	XglViewGrp3dConfig
View group interface	XglViewGrp2dItf	XglViewGrp3dItf
View concern	XglViewConcern2d	XglViewConcern3d

A view cache object consists of derived items and functions for deferred evaluation of the items. Each Context has a pointer to its own view cache object, which maintains the derived items specific to that Context.

A view group configuration object holds the static configuration information for each coordinate system from which geometry can enter LI-1. Each view cache has an array of view group configuration objects, one for each coordinate system that the view cache supports. A 2D view cache supports MC, VDC, CC, and DC. A 3D view cache supports these four as well as LC. The configuration information is static: it is invariant once initialized and is common to all view caches of a particular dimension.

A view group interface object is a pipeline's interface to the view model derived data. This object informs a pipeline when derived items have changed as a result of either the application changing a view model attribute or a pipeline changing the coordinate system from which geometry enters the next LI-1 primitive. The view group interface also maintains functions for returning the items appropriate to the current coordinate system.

A view concern object is a description of all the derived items whose changes a pipeline is concerned with. This object is a parameter of the view group interface's fast test for changes to derived items.

Each pipeline has a pointer to a view group interface object. The view group interface has functions for creating and destroying view concern objects. A pipeline may create as many view concern objects as it needs. For example, it can have one for stroke primitives and one for surface primitives. The view cache and view group configuration objects are inaccessible to pipelines so their interfaces are not described in this document; see *XGL Architecture Guide* or the appropriate header files for more information.

## *Accessing Derived Data*

The pipeline has access to member functions of the view group interface object. Each pipeline is provided with a pointer to its view group interface object by the pipeline context classes. The 2D pipeline context class, `XglPipeCtx2d`, has a member datum of type `XglViewGrp2dItf*` called `viewGrpItf`. Likewise, `XglPipeCtx3d` has a member datum of type `XglViewGrp3dItf*` called `viewGrpItf`. The constructors of `XglPipeCtx2d` and `XglPipeCtx3d` create a new view group interface object for each Context. In general, the software pipeline and device pipeline access member functions of the view group interface with `viewGrpItf->` as a prefix, as in the following example.

```
xform = viewGrpItf->getMcToDc();
```



---

**Note** – The pipeline can access transforms and view model attributes via the Context object if the pipeline never falls back on the software pipeline. If, in fact, the pipeline never uses the software pipeline, it can access the Local Model Transform using `XglTransform* mt = ctx->getModelTrans()`. Even in this case, however, it is a good idea for pipelines to use derived data so that they work consistently with other pipelines. So the preferred way to access the Local Model Transform is to use derived data, as in

```
mt = viewGrpItf->getMcToWc.
```

---

## Registration of Concerns

A device pipeline for a device that retains view model state can create view concern objects to keep track of the derived items that the pipeline is concerned about. Typically, a pipeline's concerns vary from primitive to primitive. Surfaces are more complex than stroked primitives such as polylines and markers, so a pipeline might have more concerns for surfaces. A pipeline can create its view concern objects in its constructors.

The following example shows the constructor and destructor of the 3D device pipeline for a sample pipeline (SampDp). The constructor creates view concern objects for the stroke and surface primitives. Registration of the concerns consists of two steps:

1. Define the view flag by combining bits corresponding to the derived items that the device pipeline loads into the device for a particular primitive or group of primitives.
2. Create a view concern object from the view flag.

Note that the pipeline needs only a few items from among the large selection available in derived data. This is typical for many devices. Those devices that accelerate more functionality usually need to keep track of more derived items.

```
#include "xgli/Context3d.h"
#include "xgli/DpCtx3d.h"
#include "xgli/ViewGrp3dItf.h"
#include "DpCtx3dSampDp.h"
#include "DpDevSampDp.h"

XglDpCtx3dSampDp::XglDpCtx3dSampDp(XglContext3d* ctx,
                                   XglDpDevSampDp* dp_dev) :
```

```

        XglDpCtx3d(ctx),
        XglDpCtxSampDp((XglContext*)context), dp_dev)
{
    // Define view flag for polylines and markers.
    // Xgli_view_flag_3d  XglDpCtx3dSampDp::strokeViewFlag;
    //
    strokeViewFlag.a =XGLI_VIEW_A_MC_TO_CC |
                    XGLI_VIEW_A_CC_TO_DC;
    strokeViewFlag.b = NULL;
    strokeViewFlag.c = XGLI_VIEW_C_PARALLEL_PROJ;

    // Create a view concern object for polylines and markers.
    // XglViewConcern3d*  XglDpCtx3dSampDp::strokeConcern;
    // XglViewGrp3dItf*   XglPipeCtx3d::viewGrpItf;
    //
    strokeConcern = viewGrpItf-
>createViewConcern(strokeViewFlag);

    // Define view flag for surfaces.
    // Xgli_view_flag_3d  XglDpCtx3dSampDp::surfViewFlag;
    //
    surfViewFlag.a = strokeConcernBits.a | XGLI_VIEW_A_MC_TO_WC;
    surfViewFlag.b =strokeConcernBits.b |
                    XGLI_VIEW_B_MC_TO_LC |
                    XGLI_VIEW_B_LC_TO_MC |
                    XGLI_VIEW_B_LC_TO_CC;
    surfViewFlag.c = strokeConcernBits.c |
                    XGLI_VIEW_C_LIGHTS_MC |
                    XGLI_VIEW_C_EYE_MC |
                    XGLI_VIEW_C_LIGHTS_LC |
                    XGLI_VIEW_C_EYE_LC;

    // Create a view concern object for surfaces.
    // XglViewConcern3d*  XglDpCtx3dSampDp::surfConcern;
    //
    surfConcern = viewGrpItf->createViewConcern(surfViewFlag);

    // Set this context as the last one used for rendering to the
    // device.
    // XglContext3d*   XglDpCtx3dSampDp::lastXglCtx;
    //
    lastXglCtx = ctx;

    // Assume that we last performed lighting in MC.

```

```

        // Xgli_sam_light_coord_sys
XglDpCtx3dSampDp::lastLightCoordSys;
        //
        lastLightCoordSys = SAMPDP_LIGHT_MC;
    }

XglDpCtx3dSampDp::~XglDpCtx3dSampDp( )
{
    // Destroy view concern objects.
    //
    viewGrpItf->destroyViewConcern(strokeConcern);
    viewGrpItf->destroyViewConcern(surfConcern);
}

```

### *Bit Definitions for the View Flag*

The bit definitions for the view flag have the prefixes `XGLI_VIEW_A_`, `XGLI_VIEW_B_`, and `XGLI_VIEW_C_`. The bits with the prefix `XGLI_VIEW_A_` correspond to items common to both 2D and 3D. The bits with the prefixes `XGLI_VIEW_B_` and `XGLI_VIEW_C_` are available only for 3D.

In 2D, the view flag has the type *Xgl\_usgn32*, and any combination of the bits with the prefix `XGLI_VIEW_A_` can be stored in the view flag. In 3D, the view flag has the type *Xgli\_view\_flag\_3d*:

```

typedef struct {
    Xgl_usgn32  a;           // Part "a" for XGLI_VIEW_A...
    Xgl_usgn32  b;           // Part "b" for XGLI_VIEW_B...
    Xgl_usgn32  c;           // Part "c" for XGLI_VIEW_C...
} Xgli_view_flag_3d;

```

The 3D view flag consists of three parts: a, b, and c. Any combination of bits with the prefix `XGLI_VIEW_A_` can be stored in part "a" of the view flag; likewise for `XGLI_VIEW_B_` in part "b" and for `XGLI_VIEW_C_` in part "c".

In addition to being created, a view concern can be set with a new view flag, and it can be destroyed when a pipeline no longer needs it. The 2D view group interface functions for view concerns are:

```
XglViewConcern2d*   createViewConcern (const Xgl_usgn32);  
void                setViewConcern  (XglViewConcern2d*,  
                                   const Xgl_usgn32);  
void                destroyViewConcern (XglViewConcern2d*);
```

The 3D view group interface functions for view concerns are:

```
XglViewConcern3d*   createViewConcern (const  
Xgli_view_flag_3d&);  
void                setViewConcern  (XglViewConcern3d*,  
                                   const Xgli_view_flag_3d&);  
void                destroyViewConcern (XglViewConcern3d*);
```

---

**Note** – Setting a view concern frequently is inadvisable because the process for “compiling” a view flag into a view concern is time-consuming.

---

Table 7-5 lists the bits that a pipeline can define in the view flag.

Table 7-5 Bits for the View Flag

View Flag Masks for 2D/3D Part a	View Flag Masks for 3D Part b	View Flag Masks for 3D Part c
XGLI_VIEW_A_MC_TO_DC	XGLI_VIEW_B_LC_TO_VDC	XGLI_VIEW_C_LIGHTS_MC
XGLI_VIEW_A_MC_TO_CC	XGLI_VIEW_B_VDC_TO_LC	XGLI_VIEW_C_LIGHTS_LC
XGLI_VIEW_A_CC_TO_DC	XGLI_VIEW_B_CC_TO_LC	XGLI_VIEW_C_EYE_MC
XGLI_VIEW_A_MC_TO_WC	XGLI_VIEW_B_LC_TO_DC	XGLI_VIEW_C_EYE_LC
XGLI_VIEW_A_VDC_TO_CC	XGLI_VIEW_B_MC_TO_LC	XGLI_VIEW_C_EYE_VDC
XGLI_VIEW_A_CC_TO_VDC	XGLI_VIEW_B_LC_TO_MC	XGLI_VIEW_C_EYE_CC
XGLI_VIEW_A_WC_TO_CC	XGLI_VIEW_B_LC_TO_CC	XGLI_VIEW_C_PARALLEL_PROJ
XGLI_VIEW_A_VDC_TO_DC	XGLI_VIEW_B_WC_TO_MC	XGLI_VIEW_C_MCLIP_PLANES_MC
XGLI_VIEW_A_DC_TO_VDC	XGLI_VIEW_B_WC_TO_LC	XGLI_VIEW_C_MCLIP_PLANES_LC
XGLI_VIEW_A_WC_TO_DC	XGLI_VIEW_B_LC_TO_IC	XGLI_VIEW_C_MCLIP_PLANES_CC
XGLI_VIEW_A_DC_TO_CC	XGLI_VIEW_B_IC_TO_VDC	XGLI_VIEW_C_MCLIP_PLANES_DC
XGLI_VIEW_A_MC_TO_VDC	XGLI_VIEW_B_VDC_TO_WC	XGLI_VIEW_C_DCUE_PLANES_CC
XGLI_VIEW_A_DC_TO_MC	XGLI_VIEW_B_CC_TO_WC	XGLI_VIEW_C_DCUE_PLANES_DC
XGLI_VIEW_A_MC_TO_GMC	XGLI_VIEW_B_DC_TO_LC	XGLI_VIEW_C_VIEW_CANONICAL
XGLI_VIEW_A_GMC_TO_WC	XGLI_VIEW_B_DC_TO_WC	
XGLI_VIEW_A_WC_TO_VDC	XGLI_VIEW_B_LC_TO_WC	
XGLI_VIEW_A_VCLIP_BOUNDS_VDC	XGLI_VIEW_B_CC_TO_MC	
XGLI_VIEW_A_VCLIP_BOUNDS_CC		
XGLI_VIEW_A_VCLIP_BOUNDS_DC		
XGLI_VIEW_A_VCLIP_BOUNDS_MC		
XGLI_VIEW_A_VIEWPORT_DC		

## Determining Whether Derived Items Have Changed

A device pipeline can detect changes to derived items with a sequence of tests at three levels: messages, the composite, and the individual item. In general, a device pipeline needs to know quickly when no changes have occurred so that it can proceed directly to sending geometry to the device. Accordingly, each successive level of detection involves more effort to gain more accuracy.

### Messages

Derived items can change when the application changes a view model attribute or a pipeline changes the current coordinate system. Each type of event causes a message to be sent to the device pipeline at the time of the event; notification is not deferred. The message types are

XGLI\_MSG\_VIEW\_CTX\_ATTR and XGLI\_MSG\_VIEW\_COORD\_SYS for context attribute changes and current coordinate system changes, respectively. See “Handling Derived Data Changes” on page 105 for additional information on messages. Messages of the two types above give advance warning that the next primitive may need to get derived items. A pipeline may choose to deal with the messages simply by setting its own flag at the time of the notification, then deferring action until the next primitive when it would need to interrogate the composite at the next level.

## *The Composite*

If a message reports that a change has occurred, a device pipeline can test for changes to the derived items about which it is concerned by checking the *composite*. The composite records the state changes of all derived items. The changes could be caused either by the application changing view model attributes or by a pipeline changing the current coordinate system. The composite can be thought of as all the separate derived items joined into a single unit.

## *Detecting Changes With the Composite*

The function that checks the composite has the following definition:

```
Xgl_boolean    changedComposite(const XglViewConcern{2,3}d*);
```

This function is the fast test described in “Changes to Derived Items” on page 153. The view group interface tests the composite to detect changes to derived items of concern to the device pipeline.

The view concern acts as a filter on the composite so that `changedComposite()` returns `TRUE` only when an item of interest to the pipeline has changed. If the test is `TRUE`, the pipeline needs to check each of the individual items for changes. The tests for individual items comprise the third level, and they are described in the section “Detecting Changes to Individual Derived Items” on page 166.

Recall that `changedComposite()` sometimes errs on the cautious side so that `changedComposite()` can be fast. It never misses a change in state caused by invalidation of relevant view model attributes or changes in the current coordinate system, but it may incorrectly return `TRUE` after a change to the current coordinate system. The tests at the third level for detecting changes to

individual items are fast and accurate, so extraneous reloading of view model state to a device would not occur even if `changedComposite()` incorrectly returns `TRUE`.

A device pipeline should call `changedComposite()` whenever one of its primitives regains control from the application. Typically, this is at the beginning of an LI-1 primitive. If a primitive changes the coordinate system and calls a secondary LI-1 primitive, then the original primitive should restore the original coordinate system when the secondary returns, and the original should call `changedComposite()`.

### *Setting the Composite*

The view group interface can notify a device pipeline when its concerns have changed, but it cannot detect context switches. A context switch occurs when an application renders to a device with an XGL Context after having previously rendered to the same device with a different XGL Context. If a device has only one hardware context, a context switch requires the retained state to be updated with the corresponding information of the new XGL Context. If a device has multiple hardware contexts, a device pipeline may be implemented so that an XGL Context has a one-to-one mapping with a hardware context such that a context switch does not result in reloading of retained state. Since each device handles context switches in its own way, the view group interface does not react automatically to context switches. Instead, the view group interface provides a function to set the composite:

```
void    setComposite();
```

When a context switch occurs, a device pipeline can call `setComposite()` to force the next call to `changedComposite()` and each of the tests for changes to individual items to be `TRUE`. Consequently, a device pipeline would reload its derived items into the device.

### *Clearing the Composite*

In certain situations a device pipeline may want to ignore changes to its concerns. The view group interface provides a function to clear the composite. For 2D and 3D, this function is:

```
void    clearComposite(const XglViewConcern{2,3}d*);
```

This function forces the next call to `changedComposite()` to return `FALSE` if there have been no further changes to the API view model attributes or to the current coordinate system. A pipeline may gain performance with this function because it allows primitives to ignore changes deemed to be irrelevant. But it should be used with great caution because it clears the record of inconsistencies between the state stored in the device and the actual state, which may cause a pipeline to miss a change when it becomes relevant. It should be called after `changedComposite()`.

### *Detecting Changes to Individual Derived Items*

If `changedComposite()` returns `TRUE`, a device pipeline needs to check for changes to individual items. The view group interface provides a function for each item to return the change status of that item. These functions should be called only after calling `changedComposite()`. After doing so, a pipeline may call any change function for individual items, even those that are not registered as concerns. Calling these functions does not reset the flags stored in the composite. These functions return the correct change status of individual items even when `changedComposite()` errs on the cautious side.

See the sections “Coordinate Systems and Transforms” and “Other Derived Items” for naming conventions.



Table 7-6 lists the functions to check individual items for 2D and 3D.

**Table 7-6** Functions to Return the Change Status of Derived Items

2D and 3D	3D only
Xgl_boolean changedMcToDc()	Xgl_boolean changedLctoVdc()
Xgl_boolean changedMcToCc()	Xgl_boolean changedVdcToLc()
Xgl_boolean changedCcToDc()	Xgl_boolean changedCcToLc()
Xgl_boolean changedMcToWc()	Xgl_boolean changedLcToDc()
Xgl_boolean changedVdcToCc()	Xgl_boolean changedMcToLc()
Xgl_boolean changedCcToVdc()	Xgl_boolean changedLcToMc()
Xgl_boolean changedWcToCc()	Xgl_boolean changedLcToCc()
Xgl_boolean changedVdcToDc()	Xgl_boolean changedWcToMc()
Xgl_boolean changedDcToVdc()	Xgl_boolean changedWcToLc()
Xgl_boolean changedWcToDc()	Xgl_boolean changedLcToIc()
Xgl_boolean changedDcToCc()	Xgl_boolean changedIcToVdc()
Xgl_boolean changedMcToVdc()	Xgl_boolean changedVdcToWc()
Xgl_boolean changedDcToMc()	Xgl_boolean changedCcToWc()
Xgl_boolean changedMcToGmc()	Xgl_boolean changedDcToLc()
Xgl_boolean changedGmcToWc()	Xgl_boolean changedDcToWc()
Xgl_boolean changedWcToVdc()	Xgl_boolean changedLcToWc()
Xgl_boolean changedVclipBoundsVdc()	Xgl_boolean changedCcToMc()
Xgl_boolean changedVclipBoundsCc()	Xgl_boolean changedLightsMc()
Xgl_boolean changedVclipBoundsDc()	Xgl_boolean changedLightsLc()
Xgl_boolean changedVclipBoundsMc()	Xgl_boolean changedEyeMc()
Xgl_boolean changedViewportDc()	Xgl_boolean changedEyeLc()
	Xgl_boolean changedEyeVdc()
	Xgl_boolean changedEyeCc()
	Xgl_boolean changedParallelProj()
	Xgl_boolean changedMclipPlanesMc()
	Xgl_boolean changedMclipPlanesLc()
	Xgl_boolean changedMclipPlanesCc()
	Xgl_boolean changedMclipPlanesDc()
	Xgl_boolean changedDcuePlanesCc()
	Xgl_boolean changedDcuePlanesDc()
	Xgl_boolean changedViewCanonical()

## Getting Derived Items

If an individual derived item has changed as reported by the corresponding function, a device pipeline should get the item and reload the state into the hardware. The view group interface provides a function for each item to get that item. Calling one of these functions triggers any deferred calculations that may be necessary to bring the item up to date. Therefore, a pipeline should not retain a pointer to a derived item after a primitive has finished execution because accessing the derived item with the pointer without calling the function for getting the item means that the item will not be evaluated if necessary.

The view group interface returns the requested item that is appropriate to the current coordinate system. For example, if the current coordinate system is LC and the pipeline requests the McToCc Transform, then `getMcToCc()` returns the LcToCc Transform because the geometry is in LC. A device pipeline does not need to be aware of the current coordinate system. An LI-1 primitive can be written as if geometry always enters from MC as long as it uses derived data. If a pipeline is using derived data, it must get all its Transforms from the view group interface instead of the Context. For example, a pipeline should use `viewGrpItf->getMcToGmc()` instead of `ctx->getLocalModelTrans()`. The only exception is when a pipeline wants to get the actual Transform visible at the API level with the knowledge that it may not be applicable to the current coordinate system maintained by derived data. A change in the current coordinate system is another reason that a pipeline should not retain a pointer to a derived item after a primitive has finished execution: the item returned by the view group interface may differ between primitive calls when the current coordinate system changes.

When a pipeline calls a function for getting an item, that function clears the bit in the composite that corresponds to the item. If a pipeline gets all the items that have changed, then `changedComposite()` returns `FALSE` until the pipeline's concerns change again. A pipeline can clear bits in the composite without getting changed items by calling `clearComposite()`.

Pipelines that do not retain state (such as the software pipeline) can get derived items without checking the composite or any of the individual items. While this is true of any pipeline, even those that retain state, checking the composite and individual items eliminates unnecessary loading of data into the device.

Note that if a pipeline uses derived data, it can ignore most Context view model attributes. For example, it can ignore the value of the Context attribute `XGL_CTX_VDC_MAP` because derived data takes into account the value of the VDC map when it calculates the VDC-to-DC Transform. Consequently, all Transforms derived from the VDC-to-DC Transform have the VDC mapping taken into account.

## *Getting Derived Transforms*

The view group interface allows pipelines to access numerous Transforms for mapping points forward (toward DC) and backward (toward MC); for brevity, we call these point-forward and point-backward Transforms, respectively. The point-backward Transforms can be used to map normal and direction vectors forward. Thus, the point-backward Transforms are normal-forward Transforms, and the point-forward Transforms are normal-backward Transforms.

The view cache computes the normal-forward Transforms by inverting point-forward Transforms. If an application specifies a singular<sup>1</sup> Local Model, Global Model, or View Transform, the view cache cannot compute unique normal-forward Transforms and certain derived items such as eye positions or vectors, model clip planes, and lights. Derived data currently does not claim to support singular Transforms so it is the application's responsibility to avoid singular Transforms. However, if a pipeline needs to determine if a normal-forward Transform obtained from the view group interface is valid, it should get the `McToWc`, `LcToVdc`, and `VdcToDc` Transforms after getting the normal-forward Transform and confirm that all three are nonsingular.

The view cache in 3D automatically adjusts for the effect of `XGL_3D_CTX_JITTER_OFFSET` so pipelines using derived data do not need to take this into account.

See Chapter 6, "Getting Information from XGL Objects" for information on getting data from Transform objects.

---

1. A singular matrix has no unique inverse.

Table 7-7 lists the functions for getting derived transforms for 2D and 3D.

*Table 7-7 Functions for Getting Derived Transforms*

2D and 3D	3D only
XglTransform* getMcToDc()	XglTransform* getLctoVdc()
XglTransform* getMcToCc()	XglTransform* getVdcToLc()
XglTransform* getCcToDc()	XglTransform* getCcToLc()
XglTransform* getMcToWc()	XglTransform* getLcToDc()
XglTransform* getVdcToCc()	XglTransform* getMcToLc()
XglTransform* getCcToVdc()	XglTransform* getLcToMc()
XglTransform* getWcToCc()	XglTransform* getLcToCc()
XglTransform* getVdcToDc()	XglTransform* getWcToMc()
XglTransform* getDcToVdc()	XglTransform* getWcToLc()
XglTransform* getWcToDc();	XglTransform* getLcToIc()
XglTransform* getDcToCc()	XglTransform* getIcToVdc()
XglTransform* getMcToVdc()	XglTransform* getVdcToWc()
XglTransform* getDcToMc()	XglTransform* getCcToWc()
XglTransform* getMcToGmc()	XglTransform* getDcToLc()
XglTransform* getGmcToWc()	XglTransform* getDcToWc()
XglTransform* getWcToVdc()	XglTransform* getLcToWc()
	XglTransform* getCcToMc()

## Getting Boundaries

The view group interface offers the functions listed in Table 7-8 for getting the DC viewport and the view clip bounds in MC, VDC, CC, and DC.

*Table 7-8 Functions for Getting Boundaries*

Dimension	Function
2D	const Xgl_bounds_d2d& getViewportDc() const Xgl_bounds_d2d& getVclipBoundsMc() const Xgl_bounds_d2d& getVclipBoundsVdc() const Xgl_bounds_d2d& getVclipBoundsCc() const Xgl_bounds_d2d& getVclipBoundsDc()
3D	const Xgl_bounds_d3d& getViewportDc() const Xgl_bounds_d3d& getVclipBoundsMc() const Xgl_bounds_d3d& getVclipBoundsVdc() const Xgl_bounds_d3d& getVclipBoundsCc() const Xgl_bounds_d3d& getVclipBoundsDc()

See the man page for `XGL_CTX_DC_VIEWPORT` for a description of the DC viewport. A pipeline should not use the DC viewport for view clipping; instead, it should use the view clip bounds in DC.

The view clip bounds in VDC may differ from the value of `XGL_CTX_VIEW_CLIP_BOUNDS` as specified by the application. The view cache ensures that the view clip bounds are entirely within the value of `XGL_CTX_VDC_WINDOW` in VDC and the viewport in DC. The view clip bounds in CC is always  $[-1,1] \times [-1,1]$  in 2D and  $[-1,1] \times [-1,1] \times [-1,1]$  in 3D. A pipeline should ensure that geometry never extends outside the view clip bounds.

The value of the view clip bounds in MC is the smallest rectangular parallelepiped whose edges are parallel to the coordinate axes of MC such that the parallelepiped contains the actual view clip bounds transformed into MC. This is a useful form for fast bounding-box checking in MC, but it is not particularly useful for view clipping.

If the current coordinate system is LC, `getVclipBoundsMc()` returns an incorrect value because the view cache currently has no function for evaluating the view clip bounds in LC.

## *Getting 3D Viewing Flags*

The 3D view group interface has two functions for getting more information about the WcToVdc Transform. These functions are:

```
Xgl_boolean    getParallelProj()  
Xgl_boolean    getViewCanonical()
```

A pipeline can determine if the WcToVdc Transform is configured for parallel projection by calling `getParallelProj()`, which returns `TRUE` for parallel projection and `FALSE` for perspective projection.

A pipeline can call `getViewCanonical()` to determine if the view cache successfully factored the View Transform to extract Lighting Coordinates. The function returns `TRUE` when the decomposition is successful and `FALSE` for unsuccessful. For the unsuccessful case, LC is the same as WC and IC is the same as VDC.

## Getting Lights

The 3D view group interface has two functions for getting lights in MC and LC. These functions are:

```
const XglLight* const *getLightsMc();
const XglLight* const *getLightsLc();
```

A pipeline should get the number of lights from the Context (see `XGL_3D_CTX_LIGHT_NUM(3)`) to access the array of `XglLight` pointers.

A pipeline can always perform lighting calculations in WC and LC to obtain correct results. Performing lighting calculations in MC may be faster because normal vectors do not need to be transformed, but lighting calculations in MC are correct only when the McToWc Transform preserves angles. The reason is that dot products in MC are different than those in WC when the McToWc Transform does not preserve angles. See the Transform section in Chapter 6, “Getting Information from XGL Objects” for information on how to determine whether a Transform preserves angles.

The view cache calculates the lights in MC by inverting the McToWc Transform. If the application has specified a singular matrix for the Local or Global Model Transforms, then the view cache is unable to calculate the lights in MC. A pipeline can determine if the lights in MC are valid by getting the McToWc Transform after getting the lights, then checking if it is nonsingular.

If the current coordinate system is VDC, CC, or DC, then `getLightsMc()` and `getLightsLc()` return incorrect results because the view cache currently has no functions for calculating the lights in VDC, CC, and DC. In general, lighting calculations would not be correct in these coordinate systems because the Transform from WC to VDC, CC, or DC often involves anisotropic scaling and perspective, which do not preserve angles.

See the Light section in Chapter 6, “Getting Information from XGL Objects” for information on getting data from Light objects.

## *Getting Eye Positions or Vectors*

The 3D view group interface has four functions for getting eye positions or vectors in MC, LC, VDC, and CC. These functions are:

```
const Xgl_pt_d3d&  getEyeMc()  
const Xgl_pt_d3d&  getEyeLc()  
const Xgl_pt_d3d&  getEyeVdc()  
const Xgl_pt_d3d&  getEyeCc()
```

Eye points or vectors may be used for facet orientation and lighting. Eye vectors point from eye to object along the line of sight, and the eye is located infinitely far away. Eye vectors returned by these functions have unit length.

The eyes in VDC and CC are always vectors. A pipeline can determine whether the eyes in MC and LC are positions or vectors by calling `getParallelProj()`; a parallel projection means that the eyes are vectors, while perspective means the eye are positions.

The view cache calculates eyes by inverting various Transforms. If the application has specified a singular matrix, then the view cache is unable to calculate some eyes. A pipeline can determine if the eyes in VDC and CC are valid by getting the `VdcToDc` Transform after getting the eyes, then checking if it is nonsingular (see the Transform section in Chapter 6, “Getting Information from XGL Objects”). For the eye in LC, a pipeline needs to check the `LcToVdc` and `VdcToDc` Transforms for nonsingularity. For the eye in MC, a pipeline needs to check the `McToWc`, `LcToVdc`, and `VdcToDc` Transforms for nonsingularity.

If the current coordinate system is DC, then these four functions return incorrect values because the view cache currently has no function for calculating the eye in DC. However, the value of the eye vector in DC is always (0, 0, 1).

## Getting Model Clip Planes

The 3D view group interface has four functions for getting the model clip planes in MC, LC, CC, and DC.

```
const Xgli_plane* getMclipPlanesMc()
const Xgli_plane* getMclipPlanesLc()
const Xgli_plane* getMclipPlanesCc()
const Xgli_plane* getMclipPlanesDc()
```

A pipeline should get the number of model clip planes from the Context (see `XGL_3D_CTX_MODEL_CLIP_PLANE_NUM(3)`) to access the array of *Xgli\_plane*. The structure definition is:

```
struct Xgli_plane {
    Xgl_pt_d3d    pt;
    Xgl_pt_d3d    normal;
    double        p_dot_n;
};
```

The value of *pt* is a point on the plane. The normal vectors point in the direction of accepted geometry (see `XGL_3D_CTX_MODEL_CLIP_PLANES(3)`). Normal vectors have unit length as long as the application specifies model clip planes in WC with unit normal vectors. The value of *p\_dot\_n* is the dot product of *pt* and *normal*.

The view cache calculates model clip planes by inverting various Transforms. If the application has specified a singular matrix, then the view cache will be unable to calculate some or all model clip planes. A pipeline can determine if the model clip planes in MC are valid by getting the McToWc Transform after getting the model clip planes, then checking if it is nonsingular. For the model clip planes in CC and DC, a pipeline needs to check the LcToVdc and VdcToDc Transforms for nonsingularity.

If the current coordinate system is VDC, then `getMclipPlanesMc()` and `getMclipPlanesLc()` return incorrect values because the view cache currently has no function for calculating the model clip planes in VDC.



## Getting Depth Cue Reference Planes

The 3D view group interface has two functions for getting the depth cue reference planes in CC and DC.

```
void    getDcuePlanesCc(double [])
void    getDcuePlanesDc(double [])
```

Pipelines should pass an array of 2 doubles to these functions. The value at index 0 is the front depth cue reference plane's Z-value; the value at index 1 is the back depth cue reference plane's Z-value. See

XGL\_3D\_CTX\_DEPTH\_CUE\_REF\_PLANES.

## Example of Detecting Changes and Getting Derived Items

In this example of a device pipeline for `lilTriangleStrip()`, the pipeline determines whether any Context attributes or derived items have changed by checking the flag that the pipeline sets upon receiving a message of the types `XGLI_MSG_VIEW_CTX_ATTR` or `XGLI_MSG_VIEW_COORD_SYS`. If the flag is set, the pipeline determines whether any derived items have changed by calling `viewGrpItf->changedComposite(surfConcern)`. The parameter is an `XglViewConcern3d*`, which was created in the example constructor on page 159. If `changedComposite()` indicates that derived data items have changed, the pipeline checks whether individual items have changed, and if so, it gets them from the view group interface object and loads them into the device.

You can copy or modify this source code sample as long as the resulting code is used to create a loadable pipeline for XGL.

```
#include "xgli/Context3d.h"
#include "xgli/DpCtx3d.h"
#include "xgli/Transform.h"
#include "xgli/ViewGrp3dItf.h"
#include "DpCtx3dSampDp.h"

XglDpCtx3dSampDp::lilTriangleStrip(XglPrimData* pd)
{
    // Check for context switch
    //
    if (lastXglCtx != ctx) {
        // Force reloading of attributes and derived items
        //
```

```

        udTable.setAllGroupsAsChanged();
        viewGrpItf->setComposite();
        lastXglCtx = ctx;
    }

    // Check if any view-change messages have been received
    //
    if (viewMsgReceived) {
        // Clear flag
        viewMsgReceived = FALSE;

        // Check composite for changes to surface concerns
        if (viewGrpItf->changedComposite(surfConcern)) {

            if (viewGrpItf->changedMcToCc()) {
                XglTransform*      trans;
                const Xgli_matrix_f4x4* matrix;

                trans = viewGrpItf->getMcToCc();
                matrix = (const Xgli_matrix_f4x4*)
                    trans->getMatrixFloat();

                // Write the matrix into the device
                SAMPDP_WRITE_MC_TO_CC(matrix);
            }

            if (viewGrpItf->changedCcToDc()) {
                XglTransform*      trans;
                const Xgli_matrix_f4x4* matrix;

                trans = viewGrpItf->getCcToDc();
                matrix = (const Xgli_matrix_f4x4*)
                    trans->getMatrixFloat();

                // Write the matrix into the device
                SAMPDP_WRITE_CC_TO_DC(matrix);
            }

            if (viewGrpItf->changedParallelProj()) {
                // Write the flag into the device
                SAMPDP_WRITE_PARALLEL_PROJ
                    (viewGrpItf->getParallelProj());
            }

            if (viewGrpItf->changedEyeMc()) {

```

```

        // Write the eye into the device
        SAMPDP_WRITE_EYE_MC(viewGrpItf->getEyeMc());
    }

    if (lastLightCoordSys == SAMPDP_LIGHT_MC) {
        // We performed lighting in MC last time
        if (viewGrpItf->changedMcToWc()) {

            if (viewGrpItf->getMcToWc()->getMemberRecord() &
                XGL_TRANS_GROUP_ANGLE_PRESERV) {

                // McToWc changed, but it still preserves
                // angles so we can continue to
                // perform lighting in MC.

                const XglLight* const * lights;
                lights = viewGrpItf->getLightsMc();
                // Write the lights into the device
                Xgl_usgn32 num;
                num = ctx->getLightNum();
                SAMPDP_WRITE_LIGHTS(num, lights);
            }
            else {
                // McToWc changed and it doesn't preserve
                // angles so we have to switch to performing
                // lighting in LC.

                const XglLight* const * lights;
                lights = viewGrpItf->getLightsLc();
                // Write the lights into the device
                Xgl_usgn32 num;
                num = ctx->getLightNum();
                SAMPDP_WRITE_LIGHTS(num, lights);

                // Switch lighting coordinate system
                SAMPDP_WRITE_LIGHT_COORD_SYS(SAMPDP_LIGHT_LC);
                lastLightCoordSys = SAMPDP_LIGHT_LC;
            }
        }
        else {
            // McToWc didn't change, but the lights
            // may have changed.
            //
            if (viewGrpItf->changedLightsMc()) {
                const XglLight* const * lights;

```

```

        lights = viewGrpItf->getLightsMc();

        // Write the lights into the device
        Xgl_usgn32 num;
        num = ctx->getLightNum();
        SAMPDP_WRITE_LIGHTS(num, lights);
    }
}
else {
    // We performed lighting in LC last time
    if (viewGrpItf->changedMcToWc()) {

        if (viewGrpItf->getMcToWc()->getMemberRecord() &
            XGL_TRANS_GROUP_ANGLE_PRESERV) {

            // McToWc changed and it preserves angles so
            // we can switch to performing lighting in MC.
            const XglLight* const * lights;
            lights = viewGrpItf->getLightsMc();
            // Write the lights into the device
            Xgl_usgn32 num;
            num = ctx->getLightNum();
            SAMPDP_WRITE_LIGHTS(num, lights);

            // Switch lighting coordinate system
            SAMPDP_WRITE_LIGHT_COORD_SYS(SAMPDP_LIGHT_MC);
            lastLightCoordSys = SAMPDP_LIGHT_MC;
        }
        else {
            // McToWc changed, but it still doesn't
            // preserve angles so we have to
            // continue lighting in LC.
            const XglLight* const * lights;
            lights = viewGrpItf->getLightsLc();
            // Write the lights into the device
            Xgl_usgn32 num;
            num = ctx->getLightNum();
            SAMPDP_WRITE_LIGHTS(num, lights);
        }
    }
    else {
        // McToWc didn't change, but the lights may have
        // changed.
        if (viewGrpItf->changedLightsLc()) {

```

```

const XglLight* const * lights;
lights = viewGrpItf->getLightsLc();
// Write the lights into the device
Xgl_usgn32 num;
num = ctx->getLightNum();
SAMPDP_WRITE_LIGHTS(num, lights);
}
}

if (lastLightCoordSys == SAMPDP_LIGHT_LC) {

// We have to perform lighting in LC so we need to
// write some additional items into the device.

if (viewGrpItf->changedMcToLc()) {
XglTransform*      trans;
const Xgli_matrix_f4x4* matrix;

trans = viewGrpItf->getMcToLc();
matrix = (const Xgli_matrix_f4x4*)
trans->getMatrixFloat();

// Write the matrix into the device
SAMPDP_WRITE_MC_TO_LC(matrix);
}

if (viewGrpItf->changedLcToMc()) {
XglTransform*      trans;
const Xgli_matrix_f4x4* matrix;

trans = viewGrpItf->getLcToMc();
matrix = (const Xgli_matrix_f4x4*)
trans->getMatrixFloat();

// Write the matrix into the device
SAMPDP_WRITE_LC_TO_MC(matrix);
}

if (viewGrpItf->changedLcToCc()) {
XglTransform*      trans;
const Xgli_matrix_f4x4* matrix;

trans = viewGrpItf->getLcToCc();
matrix = (const Xgli_matrix_f4x4*)

```

```

        trans->getMatrixFloat();

        // Write the matrix into the device
        SAMPDP_WRITE_LC_TO_CC(matrix);
    }

    if (viewGrpItf->changedEyeLc()) {
        // Write the eye into the device
        SAMPDP_WRITE_EYE_LC(viewGrpItf->getEyeMc());
    }
}

// Ready to send geometry to device
//
return sendLilTriangleStrip(pd);
}

```

## Current Coordinate System

A pipeline can get and set the current coordinate system. The current coordinate system is a member datum of the view cache, which maintains a stack exclusively for tracking the current coordinate systems of LI-1 primitives pending completion of execution. Pushing the current coordinate system onto the stack does not change the value of the member datum. Popping the top element from the stack changes the current coordinate system to that element; the value returned is the popped value.

The view group interface provides functions for manipulating the current coordinate system. For 2D, these functions are:

```

Xgli_li1_2d_coord_sys getCurCoordSys() const
void setCurCoordSys(Xgli_li1_2d_coord_sys)
void pushCurCoordSys()
Xgli_li1_2d_coord_sys popCurCoordSys()

```

where *Xgli\_li1\_2d\_coord\_sys* is defined as:

```

enum Xgli_li1_2d_coord_sys {
    XGLI_LI1_2D_COORD_SYS_MC = 0,
    XGLI_LI1_2D_COORD_SYS_VDC,
}

```

```
enum Xgli_li1_2d_coord_sys {
    XGLI_LI1_2D_COORD_SYS_CC,
    XGLI_LI1_2D_COORD_SYS_DC
};
```

For 3D, these functions are:

```
Xgli_li1_3d_coord_sys getCurCoordSys() const
void setCurCoordSys(Xgli_li1_3d_coord_sys)
void pushCurCoordSys()
Xgli_li1_3d_coord_sys popCurCoordSys()
```

where *Xgli\_li1\_3d\_coord\_sys* is defined as:

```
enum Xgli_li1_3d_coord_sys {
    XGLI_LI1_3D_COORD_SYS_MC = 0,
    XGLI_LI1_3D_COORD_SYS_LC,
    XGLI_LI1_3D_COORD_SYS_VDC,
    XGLI_LI1_3D_COORD_SYS_CC,
    XGLI_LI1_3D_COORD_SYS_DC
};
```

This example from the software pipeline's 2D annotation text primitive shows how a device pipeline can handle changes in the coordinate system. The software pipeline produces annotation text in VDC, so it pushes the current coordinate system (which is MC), sets the current coordinate system to VDC, calls the `li1MultiPolyline()` function (will render the strokes for the annotation text), and then pops the coordinate system to restore the previous one.


```
viewGrpItf->pushCurCoordSys();
viewGrpItf->setCurCoordSys(XGLI_LI1_2D_COORD_SYS_VDC);
itfMgr->li1MultiPolyline(&pd, FALSE, do_retained);
viewGrpItf->popCurCoordSys();
```





## *Window System Interactions*

---

8 

This chapter discusses the relationship between XGL and the window system. It includes information on the following topics:

- Discussion of the mechanism by which XGL communicates with the window system
- Scenario of how the `XglDrawable` object is created by XGL core and typically used by the device pipeline
- Overview of the functionality provided by the `XglDrawable` interfaces
- Detailed description of the `XglDrawable` interfaces



As you read this chapter, you will find it helpful to have access to the `Drawable.h` file.

## Overview of the *XglDrawable*

The *XglDrawable* object conceptualizes the sharing of a device with another entity, most often the window system, but possibly also a Memory Raster device or a Stream device. In the case of the window system, it also makes transparent to the pipeline whether it is running in an X client (using the DGA mechanism, PEXlib, or Xlib) or in a PEX server.

Because there are so many different ways to access target devices, the *XglDrawable* object was designed to encapsulate the various access mechanisms. Ideally, device pipelines do not need to be aware of the underlying mechanism. For example, a device pipeline can be used to render to an X window as a DGA client, within the server, or in a backing store.

*XglDrawable* objects are created by the XGL core in response to an `xgl_object_create()` call with a Device type, such as `XGL_WIN_RAS`. XGL creates the appropriate *XglDrawable* object, establishes a connection to the window system, creates the Device object, and links the *XglDrawable* object to the Device object. There is a one-to-one correspondence between the Device object and the *XglDrawable* object for that Device.

There are several subclasses to the *XglDrawable* object, each of which manages a different kind of target device. Table 8-1 lists these subclasses.

*Table 8-1* Drawable Subclasses

Subclass	Target Device
<i>XglDrawableDgaCView</i>	X11 window. This class encapsulates the DGA library.
<i>XglDrawableMem</i>	Memory Raster
<i>XglDrawableDgaRtn</i>	Backing-store device
<i>XglDrawableXpex</i>	PEXlib and Xlib device
<i>XglDrawableDgaBase</i>	Multibuffer in system memory
<i>XglDrawableDgaCached</i>	Multibuffer cached in hardware memory
<i>XglDrawableStream</i>	Stream device

---

**Note** – The device pipelines should interact with the `XglDrawable` object through the interfaces in `Drawable.h`, which contains the public interface for the `XglDrawable` hierarchy. Do not use the interfaces in the `XglDrawable` subclasses.

---

### *Services Provided by XglDrawable Class*

The `XglDrawable` class was designed to provide information and services to both the XGL core and the device pipeline. In particular, it provides the device pipeline with a way to get current clip lists and to lock out clip list changes while rendering is in progress.

Services provided by `XglDrawable` for the XGL core include:

- Establishing the connection with the window system and creating the `XglDrawable` object – `grabDrawable()`
- Terminating the connection with the window system and destroying the `XglDrawable` object – `unGrabDrawable()`

Services provided by the `XglDrawable` for the device pipelines include:

- Locking clip lists, thereby preventing the window system from changing them during rendering – `winLock()`
- Unlocking clip lists – `winUnLock()`
- Indicating whether clip lists have changed – `clipChanged()`
- Providing information about window geometry – `getWindowWidth()`, `getWindowHeight()`, `getWindowX()`, `getWindowY()`, `getWindowDepth()`
- Providing access to the clip list – `getMergeClipList()`

A more extensive discussion of the services provided for the pipelines by the `XglDrawable` begins on page 189.

## *A Typical Scenario of Drawable Creation and Use*

The creation of the `XglDrawable` object is handled automatically by the XGL core. The typical sequence of operations when a window raster is created is this:

1. A client (application) program maps a window and then creates an XGL API Device object.
2. The XGL core calls `XglDrawable::grabDrawable()` with the descriptor provided by the application. `grabDrawable()` uses the window descriptor information included in the request to determine the kind of Drawable required. This depends on the raster type (memory or window), the window type specified by the application, and whether the window system accepts the connection. `grabDrawable()` returns an `XglDrawable` object.
3. The XGL core calls `drawable->getPipeName()` to get the name of the appropriate rendering pipeline. If not already loaded, that pipeline is then loaded. The XGL core calls `dp_lib->getDpMgr()` to retrieve or create (if it doesn't exist) a `DpMgr` object to manage the physical device, and then calls `dp_mgr->createDpDev()` to create a `DpDev` object to manage the window. `getDpMgr()` and `createDpDev()` may call various `XglDrawable` functions to get information required for handling the device.
4. The pipeline should call `XglDrawable::setCursorRopFunc()` to register a function that will remove a software cursor from the window if necessary. Even pipelines for frame buffers with hardware cursors should call this function, as the window system may be displaying a cursor that is too big for the device's hardware cursor registers.

When the device pipeline is called on to render, it typically performs the following operations:

1. The pipeline calls `drawable->winLock()` to lock the clip lists.
2. The pipeline calls `drawable->windowIsObscured()` to determine whether the window is obscured. If `drawable->windowIsObscured()` returns `TRUE`, there is nothing to render, so the pipeline calls `drawable->winUnLock()` and returns.

3. The pipeline calls `drawable->clipChanged()` to determine whether the clip list changed since the last rendering operation. If `drawable->clipChanged()` returns `TRUE`, there is a new clip list. The pipeline proceeds as follows:
  - a. It calls `drawable->getWidth()`, `drawable->getHeight()`, `drawable->getX()`, and `drawable->getY()` to get the new window geometry.
  - b. It then calls `drawable->getMergeClipListCount()` to determine how many rectangles are in the clip list. Note that `MergeClipList` is a combination of the window system clip list and the XGL settable user clip list.
  - c. It calls `drawable->getMergeClipList()` to get the clip list. It loads this clip list into device hardware if applicable.
4. The pipeline renders to the frame buffer.
5. The pipeline calls `drawable->winUnlock()` to unlock the clip lists.

Note that after `winLock()` is called, `OpenWindows` and other applications must wait until `winUnlock()` is called before rendering to that window. For this reason, keeping a window locked for more than about 0.1 second is discouraged. Therefore, the `winLock()` and `winUnlock()` functions have been made as lightweight as possible. Holding on to a lock for more than a fraction of a second may result in poor window-system interaction; after three seconds, the window system will forcefully break the lock, which may result in incorrect rendering on the screen.

## *What You Should Know About Locking the Window*

The interface to DGA provides macros that serve to prevent the window clip list from changing during rendering. Locking the window also prevents other processes from rendering to the same window at the same time. All rendering pipelines should use the macros `WIN_LOCK()` and `WIN_UNLOCK()` (or the equivalent function calls `winLock()` and `winUnlock()`) around any operation that could alter the screen, or any time the pipeline needs a valid clip list. (The clip list may not be considered valid outside of a lock.) The pipeline uses these calls to explicitly lock and unlock the window unless the device supports concurrent access by multiple UNIX processes.

In the case of immediate-rendering hardware, a pipeline would use `winLock()` and `winUnLock()` around the actual rendering code:

```
WIN_LOCK(drawable) ;// cliplist is now valid
if( drawable->clipChanged() )
    // cliplist has changed since last lock
{
    // retrieve new cliplist from drawable
}
// render
WIN_UNLOCK(drawable) ;// done, cliplist no longer valid
```

Note that operations which do not depend on the clip list or change the contents of the screen do not need to be performed inside a lock. This can include things like changing rendering attributes and transformation matrices (except that the final viewport-to-screen coordinates transform depends on the size of the destination window, and thus must be done within a lock).

In the case of an asynchronous device (for example, a display-list device), the pipeline does not need to maintain the lock until rendering is complete. In this case, the pipeline needs only to hold the lock until the host has completed its access to the device. It is the responsibility of the window system and the hardware device to set up whatever synchronization protocol allows coherent rendering between them. This synchronization protocol, which is independent of XGL, most often relies on the window system requesting the accelerator to flush all its pending operations.

On a display-list device, the rendering code would look something like this:

```
WIN_LOCK(drawable) ;// cliplist is now valid and stable
if( drawable->clipChanged() )
    // clip list has changed since last lock
{
    // retrieve new cliplist from drawable
    // and download to device.
}

// download display list to device.
// initiate rendering.
WIN_UNLOCK(drawable) ;// done

// Window system will maintain stable clip list until
// rendering is complete.
```

See “Accessing Dynamic Information Through the Drawable” on page 191 for more information on locking and unlocking the window at rendering time.

## *Drawable Interfaces for the Pipeline*

The `XglDrawable` object provides a number of interfaces that allow you to:

- Obtain information about the frame buffer or the window
- Access dynamic information, such as window dimensions
- Manage window system resources

These general categories of functions are discussed in the sections that follow. For detailed descriptions of the `XglDrawable` pipeline interfaces, see page 197.

## Obtaining Information During Pipeline Initialization

Several XglDrawable functions allow you to get information that you may need about the frame buffer during device creation. Table 8-2 lists these functions.

*Table 8-2* Drawable Interfaces Used During Pipeline Initialization

Function	Description
<code>getDevFd()</code>	Returns the device file descriptor.
<code>getDeviceName()</code>	Returns a pointer to the frame buffer device name, such as <code>/dev/fb</code> .
<code>getWindowDepth()</code>	Returns the depth of the window.

For example, as part of your `getDpMgr()` function, you will probably first want to determine whether an `XglDpMgr` object for this frame buffer has already been created. One way of doing this is provided in the utility functions of the `XglListOfDpMgr` class. This class provides two functions: one to retrieve an existing `DpMgr` matching a file descriptor, and another to create a new `DpMgr` for the device.

```
XglDpMgr* XglListOfDpMgr::getDpMgr(int fildes)
void XglListOfDpMgr::addDpMgr(int fildes, XglDpMgr* mgr)
```

The code fragment below shows an example of how the pipeline can use these utility functions.

```
XglDpMgr* XglDpLibSampDp::getDpMgr(Xgl_obj_type,
                                   XglDrawable* drawable)
{
    XglDpMgr*    dpMgr;

    dpMgr = dpMgrList.getDpMgr(drawable->getDevFd());
    if (dpMgr == NULL) {
        dpMgr = new XglDpMgrSampDp(drawable->getDevFd());

        dpMgrList.addDpMgr(drawable->getDevFd(), dpMgr);
    }
    return dpMgr;
}
```



## *Accessing Dynamic Information Through the Drawable*

The primary service provided by the Drawable is to provide a mechanism to lock the window clip list during rendering. Since the window system updates the clip list and other window attributes in response to changes in the window, the XglDrawable object synchronizes access to the window information via a lock and release mechanism. Once the coordination between the client (XGL) and the server has been established, the client can draw directly to the window using the lock and release routines. Since the server can continue to update the window in response to changes in the window's characteristics, the client must lock the window clip list before drawing and unlock it when drawing is complete.

The lock function does the following:

- Locks the clip list so that the server cannot change it during a rendering operation
- Examines the clip list to see if it has changed since the last lock, and, if it has changed, the function updates the global copy of the clip list
- Merges the system clip list and the user clip list

The unlock function releases the lock on the clip list. At that point, the server can change the window at any time, and the clip lists are invalid until the next lock.

There are three kinds of clip lists that the XglDrawable object manages:

- Window system clip list. This is the clip list that is set by the window system.
- User clip list. This is the clip list that is set by the XGL application.
- Merged clip list. This clip list is obtained when the lock function merges the window system clip list and the user clip list.

Most device pipelines should use the merged clip list at all times. However, devices on which the window system sets up hardware window clipping in advance should use the user clip list.

### *Guidelines for Using the Window Lock Macros or Function Calls*

To lock and unlock the window, the `XglDrawable` object provides the pipeline with a pair of macros and a pair of function calls.

	<b>Lock</b>	<b>Unlock</b>
<b>Inline macros</b>	<code>WIN_LOCK(drawable)</code>	<code>WIN_UNLOCK(drawable)</code>
<b>Function calls</b>	<code>drawable-&gt;winLock()</code>	<code>drawable-&gt;winUnlock()</code>

If performance is an issue, you should use the `WIN_LOCK(drawable)` inline macro to lock the window, and after rendering, use the `WIN_UNLOCK(drawable)` macro to unlock the window. Both `WIN_LOCK()` and `WIN_UNLOCK()` are designed to be as lightweight as possible; in other words, no function calls are made unless the window has changed.

If performance is not critical, the `drawable->winLock()` and `drawable->winUnlock()` inline functions can be used instead. These result in function calls for `XglDrawable` objects that actually need locking, so they are not quite as lightweight as the macros but result in less generated code.

### *Between Lock and Unlock Calls*

All rendering occurs between lock and unlock calls. In addition to rendering during locks, the device pipeline may need other information, such as the current dimensions of the window. Once the window is locked, you can ask the `XglDrawable` object for the information that you need. In general, any hardware access that depends on the state of the window should be bracketed by lock and unlock calls.

The following code example shows a pipeline checking the state of the window and the status of the clip list. The clip list changes when the window moves, changes size, or is partially covered.

```
*drawable = device->getDrawable() ;

WIN_LOCK(drawable) ;
if( drawable->windowIsObscured() ) {
    //window is covered or closed
    WIN_UNLOCK(drawable) ;
    return 1 ;    // window is obscured; don't render
}

if( drawable->clipChanged() )
{
    // load new clip list into hardware
    // recompute view transformation matrices
}
// render
WIN_UNLOCK(drawable);
```

Table 8-3 lists functions that are only meaningful inside lock and unlock calls because, in general, the information that they return is valid only when the window information is locked.

**Table 8-3** Drawable Interfaces Used During Rendering

Function	Description
clipChanged()	Returns TRUE if the clip list has changed since the last time this function was called.
getClipMask()	Returns the clip mask.
getClipStat()	Returns one of DGA_VIS_UNOBSCURED, DGA_VIS_PARTIALLY_OBSCURED, or DGA_VIS_FULLY_OBSCURED. DGA_VIS_UNOBSCURED means that the window is completely exposed. DGA_VIS_PARTIALLY_OBSCURED means the window is partially clipped. DGA_VIS_FULLY_OBSCURED means that the window is completely hidden.
getMergeClipList()	Returns the clip list.

**Table 8-3** Drawable Interfaces Used During Rendering *(Continued)*

Function	Description
<code>getMergeClipListCount()</code>	Returns the number of <i>Xgl_irect</i> structures in the clip list.
<code>getWindowDepth()</code>	Returns the depth of the window.
<code>getWindowWidth()</code> <code>getWindowHeight()</code>	Return the height or width of the window.
<code>getWindowX()</code> <code>getWindowY()</code>	Return coordinates of the window.
<code>getWsClipList()</code>	Returns the window clip list.
<code>getWsClipListCount()</code>	Returns the number of <i>Xgl_irect</i> structures in the window clip list.
<code>windowIsClipped()</code>	Returns TRUE if the window is partially clipped.
<code>windowIsFullyExposed()</code>	Returns TRUE if the window is completely exposed.
<code>windowIsObscured()</code>	Returns TRUE if the window is completely obscured.

## *Xpex and Memory Raster Pipelines*

Note that for some drawable types, such as `XglDrawableDgaRtn` and `XglDrawableMem`, the concept of window locking has no meaning. However, in most cases the pipeline should call these functions as described anyway. Clip list inquiry functions will simply return the user's clip list.

## Managing Window System Resources

Some frame buffers have special characteristics, such as hardware double buffering, Z-buffers, or stereo imaging. These attributes are a limited resource and are assigned by the window system. Table 8-4 lists functions that you can use to manage resources.

**Table 8-4** Drawable Interfaces Used for Allocating Resources

Function	Description
<code>grabWids()</code>	Returns a block of window IDs from the server. Use with <code>getWid()</code> to return the IDs just allocated.
<code>grabZbuf()</code>	Communicates to the server a client request for a Z-buffer.
<code>grabFCS()</code>	Requests to allocate fast clear plane set.
<code>grabStereo()</code>	Requests stereo planes.
<code>dbGrab()</code>	Requests double buffering on the drawable.
<code>dbUnGrab()</code>	Terminates double buffering on the drawable.
<code>getWid()</code>	Returns the window IDs for the window, if applicable.
<code>setWriteBuffer()</code>	Sets the buffer to be written.
<code>setReadBuffer()</code>	Sets the buffer to be read.
<code>setDisplayBuffer()</code>	Sets the buffer to be displayed.
<code>dbDisplayComplete()</code>	Called after <code>setDisplayBuffer()</code> ; returns 1 if the new buffer is now visible.
<code>dbDisplayWait()</code>	Waits for the double-buffering interval (one frame) to expire.
<code>dbGetWid()</code>	Returns the window ID for the double-buffering window.

As an example, during your device initialization, you may want to request a Z-buffer and specify hardware double buffering, since your hardware supports multiple buffers. A minimal implementation of these calls might be:

```
XglDrawable* drawable = device->getDrawable();
if (!drawable->grabZbuf(1)) { //request the Z buffer
    return error;
}
if (drawable->dbGrab(2, (void(*)())vrtfunc, cpage)
    { //request double buffering
        //set up hardware
    } else { //server didn't comply with request
        return 1;
    }
```

When the device pipeline is using double buffering, it is the pipeline's responsibility to inform the server/DGA of the buffer switch. To do this, use the relevant XglDrawable functions. See page 197 for a more complete description of the XglDrawable interfaces.

## *Managing Software Cursors*

For frame buffers with software cursors, the XglDrawable object must be able to erase the cursor before drawing. The `setCursorRopFunc()` passes the Drawable a pointer to a device pipeline function that erases the cursor whenever necessary. Although XGL does not include a user-defined cursor, the pipeline should define the `setCursorRopFunc()` so that DGA can call it to copy the image under the software cursor (as passed in by a parameter to the cursor rop function) when the cursor is on top of the display window.

## *Description of Drawable Interfaces*

The following is an alphabetized list of the XglDrawable operators. This list provides the syntax and description for each function. It also provides you with hints about how you can best optimize XglDrawable accesses within a pipeline. The hints are in the form of the following codes:

- [E]      The function is time consuming to call; in other words, the subroutine call has many tasks to perform.
- [M]      The function is moderately time-consuming; the subroutine call does very little.
- [L]      The function is lightweight because it is inline code.
- [L2]     The function is basically a lightweight function that is only time consuming if there has been a clip list change.

---

**Note** – The XglDrawable interface and any DGA interfaces mentioned in this chapter are uncommitted and subject to change.

---

## *XglDrawable Functions for the Device Pipeline*

```
void winLock()
```

This function locks the raster's clip list and other information in the shared memory data structure, making it possible to render. All rendering must be between `winLock()` and `winUnlock()` calls.

This is an inline function for efficiency. In the noncontested case, it is very fast. `winLock()` and `winUnlock()` calls should be run fairly frequently so that the cursor and other updates on the screen are fast. Under no circumstances should XGL hold onto a lock for more than three seconds, since this can cause a timeout. [L2]

```
void winUnlock()
```

Unlock the shared-memory data structure. [L2]

```
WIN_LOCK(d)
```

Lock the window. This macro is more efficient than using `winLock()`, but it expands to more code. [L2]

`WIN_UNLOCK(d)`

Unlock the window. [L2]

`Xgl_boolean clipChanged()`

Returns `TRUE` if the clip list has changed since the last time this function was called. Only valid inside a lock. [L]

`int dbDisplayComplete(int waitflag)`

Returns 1 if the new displayed buffer is now visible. If the new buffer is not yet displayed, and `waitflag` is zero, returns 0. If `waitflag` is set, `dbDisplayComplete()` waits for the display to be visible if necessary and always returns 1. [E]

`void dbDisplayWait()`

Waits for the double-buffering interval (one frame) to expire. [E]

`u_int dbGetWid()`

Returns the window ID for the double-buffering window. Meaningful only for frame buffers that use window IDs for double buffering. See also “Window System Dependencies”. [M]

`Xgli_ClipStat getClipStat()`

Returns one of `DRW_EXPOSED`, `DRW_CLIPPED`, `DRW_OBSCURED`. Only valid inside a lock. [L]

`int getDevFd()`

Returns the device file descriptor for the frame buffer on which the grabbed window is displayed. [M]

`XglDevice* getDevice()`

Returns the back pointer to the corresponding Device object, which may be `XglRasterWin`, `XglRasterMem`, and so on.

`const char * getDeviceName()`

Returns a pointer to the device name of the frame buffer on which the grabbed window is displayed, for example `/dev/cgsix0`. Note that the device has already been opened. [M]



```
const Xgl_irect_list& getMergeClipList()
```

Returns the clip list. Only valid inside a lock. [L2]

```
Xgl_sgn32 getMergeClipListCount()
```

Returns the number of *Xgl\_irect* structures in the clip list. Only valid inside a lock. [L2]

```
XglPixRectMem* getMergeClipMask()
```

Returns a bitmap representing the visible portion of the window.

```
Xgl_color_type getRealColorType()
```

Returns the actual color type of the underlying hardware, which can be one of `XGL_COLOR_INDEX` or `XGL_COLOR_RGB`.

```
void getWid(int &nwid, int &start_wid, int &cur_wid)
```

Returns the window IDs for the window, if applicable. `nwid` is the number of window IDs, `start_wid` is the first window ID, and `cur_wid` is the current window ID. [M]

```
Xgl_sgn32 getWindowDepth()
```

Get window depth. [E]

```
Xgl_sgn32 getWindowWidth()
```

```
Xgl_sgn32 getWindowHeight()
```

Return overall window geometry, including parts that may be clipped. Only valid inside a lock. [L]

```
Xgl_sgn32 getWindowX()
```

```
Xgl_sgn32 getWindowY()
```

Return overall window geometry, including parts that may be clipped. Only valid inside a lock. [L]

```
Xgl_sgn32 getWsClipListCount()
```

Returns the number of *Xgl\_irect* structures in the window clip list. Only valid inside a lock. [L]

```
const Xgl_irect_list& getWsClipList()
```

Returns the window clip list. Only valid inside a lock. [L]

```
Xgl_sgn32 getUserClipListCount()
```

Returns the number of *Xgl\_irect* structures in the user clip list. [L]

```
const Xgl_irect_list& getUserClipList()
```

Returns the user clip list. [L]

```
Xgl_boolean dbGrab(int nbuffers,
    void(*vrtfunc)(Dga_window), u_int* vrtcounterp)
```

Requests double buffering on this Drawable with *nbuffers*. Both *vrtfunc* and *vrtcounterp* are supplied by the device pipeline. For more information on the implementation of this function, see *dga\_db\_grab()* in the *OpenWindows Server Device Developer's Guide*. Returns *TRUE* for success and *FALSE* for failure. [E]

```
Xgl_boolean grabFCS(int nfcs)
```

Grabs *nfcs* fast clear sets. Releases fast clear sets by setting *nfcs* to zero. Returns *FALSE* for failure and *TRUE* for success. Currently only succeeds for OpenWindows windows and only when supported by the hardware. Fast clear set information is stored in a device-dependent manner. See “Window System Dependencies”. [E]

```
Xgl_boolean grabWids(int nwids)
```

Grabs *nwids* window IDs. Returns *FALSE* on failure. [E]

```
Xgl_boolean grabZbuf(int nzbuftype)
```

Grabs or releases the Z-buffer where 1 means grab and 0 means release. Returns *FALSE* for failure, *TRUE* for success. Currently only succeeds for OpenWindows windows and only when supported by hardware. Z-buffer information is stored in a device-dependent manner. See “Window System Dependencies”. [E]

```
Xgl_boolean grabStereo(int st_mode)
```

Grab or release the stereo planes; 1 means grab, 0 means release. Returns FALSE for failure, TRUE for success. Currently only succeeds for OpenWindows windows and only when supported by hardware. Stereo plane information is stored in an undocumented device-dependent manner. See “Window System Dependencies”. [E]

```
void setCursorRopFunc(void * my_rop_func, caddr_t client)
```

Sets the function that is used to remove the cursor from the screen. `my_rop_func` is a function provided by the pipeline. This function is called by DGA to copy the image under the software cursor as passed in through the `caddr_t memptr` parameter to the cursor rop function when the cursor is on top of the display window. The function should look like this :

```
void
my_rop_func(XglDevice *dev, int x, int y, int width, int height,
            int depth, int linebytes, caddr_t memptr,
            caddr_t client)
```

This function is called from within `WIN_LOCK()` whenever the cursor needs to be taken down. Its purpose is to copy a block of pixels onto the frame buffer, thus undrawing the cursor. The `dev` pointer is the XGL Device of the window for which the cursor is being undrawn; to retrieve the `DpDev`, get the `XglDpDev` object with `device->getDpDev()`. The arguments `x,y,w,h,depth` describe the region of the screen to be replaced. `linebytes` and `memptr` describe the source for the pixels. `client` is the arbitrary client data provided to `setCursorRopFunc()`. `memptr` points to the (0,0) pixel address of the image. The format is a row-column order with each row starting `linebytes` after the previous row. Note that no XGL attribute (that is the ROP and the plane mask) is relevant within this function.

All pipelines should provide this function if it is at all possible for a software cursor to intersect this drawable. [M]

```
void setDisplayBuffer(int buffer, int (*displayfunc)(),
                     caddr_t data)
```

Sets the buffer to be displayed. `displayfunc` is a function that you provide in the form:

```
int displayfunc(caddr_t data, Dga_window clientp, int buffer)
```

where `data` is the data provided, `clientp` is the client info pointer described in the OpenWindows DDK documentation, and `buffer` is the buffer to be written. Your `displayfunc` function is device dependent and is responsible for setting the hardware to display to the specified buffer. [M]

```
void setReadBuffer(int buffer, int (*readfunc)(),
                  caddr_t data)
```

Sets the buffer to be read. `readfunc` is a function that you provide in the form:

```
int readfunc(caddr_t data, Dga_window clientp, int buffer)
```

where `data` is the data provided, `clientp` is the client info pointer described in the OpenWindows DDK documentation, and `buffer` is the buffer to be written. Your `readfunc` function is device-dependent and is responsible for setting the hardware to read from the specified buffer. [M]

```
void setWriteBuffer(int buffer, int (*writefunc)(),
                   caddr_t data)
```

Sets the buffer to be written. `writefunc` is a function that you provide in the form:

```
int writefunc(caddr_t data, Dga_window clientp, int buffer)
```

where `data` is the data provided, `clientp` is the client info pointer described in the OpenWindows DDK documentation, and `buffer` is the buffer to be written. Your `writefunc` function is device-dependent and is responsible for setting the hardware to write to the specified buffer. [M]

```
Xgl_boolean windowIsClipped()
```

Returns TRUE if window is partially exposed. Only valid inside a lock. [L]

```
Xgl_boolean windowIsFullyExposed()
```

Returns TRUE if window is completely exposed. Only valid inside a lock. [L]

```
Xgl_boolean windowIsObscured()
```

Returns TRUE if window is completely obscured. Only valid inside a lock. Data does not need to be sent to the hardware if `windowIsObscured()` is TRUE. If backing store is enabled and handled by the device pipeline, the

pipeline should check the X window system's backing store attribute to determine whether it is `WhenMapped` or `Always` to decide whether to render to the backing store if `windowIsObscured()` is `TRUE`. [L]

`Xgl_boolean dbUnGrab()`

Terminates double buffering on this drawable. Returns `TRUE` on success, `FALSE` on failure. [E]

### *XglDrawable Functions Used by the XGL Core Only*

`Xgli_DrawClass getClass()`

Returns one of `DRW_WIN_RAS`, `DRW_MEM_RAS`, or `DRW_CGM`. These identify the kind of raster that this Drawable was created for. [L]

`void getDescriptor(void *)`

Returns the original descriptor passed to `xgl_object_create()`. [M]

`DrawableLockType getLockType()`

This function is not normally relevant to device pipelines. It describes what action will be taken by `winLock()`, which can be one of `DR_LK_NONE`, `DR_LK_FUNC`, or `DR_LK_MACRO`.

`const char* getPipeName()`

Used by the XGL core to determine the proper rendering pipeline for this window.

`Xgl_window_type getType()`

Returns the `Xgl_window_type` from `xgl.h` for `DRW_WIN_RAS`. [L]

`Xgl_boolean grabRetainedWindow()`

Grabs a window for backing store. Returns an `XglDrawable` object on success and connects the new object to the existing `XglDrawable` object. Returns `NULL` on failure. Note that the retained window is actually a file in `/tmp`.

```
static XglDrawable *grabDrawable(Xgl_obj_desc *,
                                Xgl_device *)
```

Grabs the window. Returns an XglDrawable object on success, NULL on failure. Initializes most of the fields in the XglDrawableClient object. [E]

```
Xgl_boolean matchDesc(Xgl_obj_desc *)
```

Returns TRUE if the given descriptor matches this XglDrawable object. [E]

```
Xgl_boolean possible(Xgl_X_window *)
```

Determines whether DGA is possible on this window. If DGA is possible, the function returns TRUE. If DGA is not possible, returns FALSE. In the latter case, PEXlib or Xlib must be used for rendering. [E]

```
void resize()
```

Used to inform the XglDrawable object that the window has been resized. Note that this function is used only by the XglDrawableXpex subclass, since it has no other way of determining whether the window has been resized.

```
void setRectList(Xgl_irect rect_list[])
```

```
void setRectNum(Xgl_usgn32)
```

Sets the user clip list. [E]

```
Xgl_boolean unGrabRetainedWindow()
```

Terminates access to the backing-store window. The XglDrawable object and its resources are freed.

```
void ungrabDrawable()
```

Used by the XGL core to terminate access to a window. The XglDrawable and all of its resources are freed.

## *Window System Dependencies*

Unfortunately, some DGA information, such as fast clear sets, is not formally defined in OpenWindows DGA. Currently, the information is simply stored in the OpenWindows DGA shared page in a device-dependent manner.

Pipelines that need access to the double-buffering information or the bounding-box information in shared memory should use the following functions:

```
caddr_t winBboxinfop()
```

Returns a pointer to the bounding-box information structure within shared memory. This structure is:

```
struct {  
    int    xleft, xtop;  
    int    width, height;  
}
```

Returns NULL if not running under OpenWindows. [L]

```
Dga_dbinfo *winDbInfop()
```


Returns a pointer to the double-buffering information area within shared memory, as defined in `<dga/dga.h>`. Returns NULL if not running under OpenWindows. [L]





## Writing Loadable Interfaces

---

9 

This chapter describes the XGL loadable interfaces. Each interface description includes information about a function's syntax, arguments, and attributes, and the software pipeline implementation. Also provided are the mappings of API functions to LI-1 functions and the mappings of LI functions at one level to those at another level.



As you read this chapter, you will find it helpful to have access to the following header files:

- `XglDpCtx2d.h` and `XglDpCtx3d.h`. These files contain the loadable interfaces for the device pipeline.
- `XglSwpCtx2d.h` and `XglSwpCtx3d.h`. These files contain the loadable interfaces for the software pipeline.
- `Li3Structs.h`, `Li3Structs2d.h`, and `Li3Structs3d.h`, and `RefDpCtx.h`, `RefDpCtx2d.h`, and `RefDpCtx3d.h` contain information on LI-3 functions and data structures.

---

**Note** – The interfaces mentioned in this chapter are unstable and subject to change.

---

## *What You Need to Know about the Loadable Interfaces*

The XGL architecture provides considerable flexibility in implementing a device pipeline. You can implement pipelines at the LI-1 level for every XGL primitive, or you can choose to implement some primitives at the LI-1 level and some at the LI-2 level and use the software pipeline for the remaining primitives. You can also use the software pipeline for LI-1 processing, and implement some of the LI-2 interfaces and the LI-3 interfaces. Your decision will depend on the capabilities of your hardware.

The software pipeline includes a set of support routines that can fill in functionality that a pipeline cannot handle. It is likely that even pipeline ports that fully accelerate most XGL functionality will fall back to the software pipeline for some features. In general, the software pipeline will be used for devices that:

- Accelerate most XGL functionality, including transformations, clipping, lighting, and scan conversion, but may not implement particular primitives or support some combinations of XGL attributes. Device pipelines for these devices can fall back to the software pipeline for unimplemented primitives or unaccelerated features.
- Partially accelerate functionality at a particular level, such as Xlib devices or simple color frame buffer devices. For example, an Xlib device may render vectors and polygons using Xlib routines. This pipeline will use the XGL software pipeline for LI-1 operations such as transformations, clipping, lighting, and depth cueing, and will perform scan conversion at the LI-2 level in the server after the device pipeline issues an Xlib call. A simple color frame buffer may use the software pipeline LI-1 and LI-2 functions and implement only LI-3 functions to write pixel values into the hardware.

The software pipeline provides a generic implementation of what is expected of the device pipeline at each loadable interface level. By examining the software pipeline implementation, you can get a global view of what the device pipeline needs to do.

## Overview List of Loadable Pipeline Interfaces

Table 9-1 lists the set of interfaces for the device pipeline. Some of these interfaces must be implemented by every device pipeline; others are optional. The table describes the interfaces at each layer and shows whether the interfaces are implemented by the software pipeline. Note that functions that require direct pixel access or immediate interaction with the device pipeline are not simulated in software.

Table 9-1 List of Loadable Pipeline Interfaces

LI	Function	2D	3D	Description	Swp	Dp
LI-1	<code>lilAnnotationText()</code>	✓	✓	Renders text in a plane parallel to the display surface.	✓	Optional
	<code>lilDisplayGcache()</code>	✓	✓	Displays the contents of the Gcache object.	✓	Optional
	<code>lilMultiArc()</code>	✓	✓	Renders a set of arcs.	✓	Optional
	<code>lilMultiCircle()</code>	✓	✓	Renders a set of circles.	✓	Optional
	<code>lilMultiEllipticalArc()</code>		✓	Renders a set of 3D elliptical arcs.	✓	Optional
	<code>lilMultiMarker()</code>	✓	✓	Renders a set of markers.	✓	Optional
	<code>lilMultiPolyline()</code>	✓	✓	Renders a set of polylines.	✓	Optional
	<code>lilMultiRectangle()</code>	✓	✓	Renders a set of rectangles.	✓	Optional
	<code>lilMultiSimplePolygon()</code>	✓	✓	Renders a set of single-bounded polygons.	✓	Optional
	<code>lilNurbsCurve()</code>	✓	✓	Renders a NURBS curve.	✓	Optional
	<code>lilNurbsSurf()</code>		✓	Renders a NURBS surface.	✓	Optional
	<code>lilPolygon()</code>	✓	✓	Renders a single planar polygon.	✓	Optional
	<code>lilQuadrilateralMesh()</code>		✓	Renders a set of connected quadrilateral polygons.	✓	Optional
	<code>lilStrokeText()</code>	✓	✓	Renders stroke text.	✓	Optional
	<code>lilTriangleList()</code>		✓	Renders a set of triangles arranged as a triangle strip, a triangle star, or unconnected triangles.	✓	Optional
	<code>lilTriangleStrip()</code>		✓	Renders a set of connected triangular polygons.	✓	Optional

**Table 9-1** List of Loadable Pipeline Interfaces (Continued)

LI	Function	2D	3D	Description	Swp	Dp
	<code>li1Accumulate()</code>		✓	Accumulates images from the draw buffer of the raster to a specified accumulation buffer.	✓	Optional
	<code>li1ClearAccumulation()</code>		✓	Clears the accumulation buffer.	✓	Optional
	<code>li1CopyBuffer()</code>	✓	✓	Copies a block of pixels from one buffer to another.		Required
	<code>li1Flush()</code>	✓	✓	Causes pending processing to complete.		Required
	<code>li1GetPixel()</code>	✓	✓	Gets the color value of a pixel.		Required
	<code>li1Image()</code>	✓	✓	Displays a block of pixels.	✓	Optional
	<code>li1NewFrame()</code>	✓	✓	Clears the DC viewport to the background color.		Required
	<code>li1PickBufferFlush()</code>	✓	✓	Synchronizes the device's pick buffer and the XGL core pick buffer.		Required
	<code>li1SetMultiPixel()</code>	✓	✓	Sets the color values for a list of pixels.	✓	Optional
	<code>li1SetPixel()</code>	✓	✓	Sets the color value of a specified pixel.		Required
	<code>li1SetPixelRow()</code>	✓	✓	Sets the color value for a row of pixels.	✓	Optional
LI-2	<code>li2GeneralPolygon()</code>	✓	✓	Scan converts polygons to span lines.	✓	Optional
	<code>li2MultiDot()</code>	✓	✓	Calls <code>li3MultiDot()</code> .	✓	Optional
	<code>li2MultiEllipse()</code>	✓		Scan converts ellipses to span lines.	✓	Optional
	<code>li2MultiEllipticalArc()</code>	✓		Scan converts elliptical arcs to span lines.	✓	Optional
	<code>li2MultiPolyline()</code>	✓	✓	For thin lines, calls <code>li3Vector()</code> ; scan converts wide lines to span lines.	✓	Optional
	<code>li2MultiRect()</code>	✓		Scan converts rectangles to span lines.	✓	Optional
	<code>li2MultiSimplePolygon()</code>	✓	✓	Scan converts polygons to span lines.	✓	Optional
	<code>li2TriangleList()</code>		✓	Takes a triangle list, breaks it into individual triangles and scan converts the triangles.	✓	Optional
	<code>li2TriangleStrip()</code>		✓	Takes a triangle list, breaks it into individual triangles and scan converts the triangles.	✓	Optional

Table 9-1 List of Loadable Pipeline Interfaces (Continued)

LI	Function	2D	3D	Description	Swp	Dp
LI-3	li3Begin()	✓	✓	Specifies the beginning of a sequence of LI-3 primitives.		Required
	li3End()	✓	✓	Specifies the end of a sequence of LI-3 primitives.		Required
	li3CopyFromDpBuffer()	✓	✓	Copies pixel data from a hardware buffer into memory.		Required
	li3CopyToDpBuffer()	✓	✓	Copies pixel data from memory into the specified hardware buffer.		Required
	li3MultiDot()	✓	✓	Draws a list of dots at a specified x,y location.		Required
	li3MultiSpan()	✓	✓	Draws a list of spans.		Required
	li3Vector()	✓	✓	Draws a vector.		Required
	li3GetDotControl()		✓	Get control functions for LI-3 dots, vectors, and spans.		Required
	li3GetVectorControl()	✓	✓			
	li3GetSpanControl()	✓	✓			
	li3SetDotControl()		✓	Set control functions for LI-3 dots, vectors, and spans.		Required
	li3SetVectorControl()	✓	✓			
	li3SetSpanControl()	✓	✓			

### Deciding Which Interfaces to Implement

Table 9-2 shows the LI-1, LI-2, and LI-3 functions that are called by the software pipeline LI-1 functions. If you decide to implement one of the interfaces listed in the left column, you may also want to implement some or all of the checked functions listed to the right. You can think of the functions to the right as being *downstream* from the LI-1 function that calls them. In this

table, “D” indicates a function that the pipeline calls directly; “I” indicates to a function that is called indirectly by a function downstream from the LI-1 function.

Table 9-2 LI1 to LI2 Dependencies

	li1MultiPolyline	li1Polygon	li1MultiSimplePolygon	li1MultiMarker	li1TriangleList	li1TriangleStrip	li1QuadrilateralMesh	li2MultiPolyline	li2MultiDot	li2MultiSimplePolygon	li2GeneralPolygon	li2MultiEllipticalArc	li2MultiEllipse	li2TriangleList	li2TriangleStrip	li3MultiSpan	li3MultiDot	li3Vector
li1AnnotationText - 2D/3D	D							I										I
li1MultiArc - 2D	D	D						I			I	D				I		I
li1MultiArc - 3D	D	I	D					D			D					I		I
li1MultiCircle - 2D		D									I		D			I		
li1MultiCircle - 3D		I	D								D					I		
li1MultiEllipticalArc - 3D	D	I	D					D			D					I		I
li1MultiMarker - 2D/3D	D							I	D							I	I	I
li1MultiPolyline - 2D/3D								D								I		I
li1MultiRectangle - 2D		I	D								I					I		
li1MultiRectangle - 3D		I	D								D					I		
li1MultiSimplePolygon - 2D		D									I					I		
li1MultiSimplePolygon - 3D		D								D	I					I		
li1NurbsCurve - 2D/3D	D			D				I	I							I		I
li1NurbsSurf - 3D	D			D	D		D	I	I	I				I		I		I
li1Polygon - 2D/3D											D					I		I
li1QuadrilateralMesh - 3D						D								I		I		
li1StrokeText - 2D/3D	D							I								I		I
li1TriangleList - 3D											D			D	D	I		
li1TriangleStrip - 3D											D				D	I		

Note that, depending on the geometry in the Gcache, 2D `li1DisplayGcache()` calls the following LI-1 primitives:

- `li1Marker()`
- `li1MultiPolyline()`
- `li1MultiSimplePolygon()`
- `li1Polygon()`
- `li1NurbsCurve()`

For 3D, `li1DisplayGcache()` calls all of the above and

- `li1NurbsSurf()`
- `li1TriangleStrip()`.

Table 9-3 shows which LI-2 and LI-3 functions are called by each of the LI-2 functions. If you decide to replace one of the functions listed in the left column, you may also want to replace the functions listed to the right.

*Table 9-3* LI-2 to LI-3 Dependencies

	li2MultiPolyline	li2GeneralPolygon	li2TriangleList	li2MultiSimplePolygon	li3Multispan	li3Multidot	li3Vector
<b>li2GeneralPolygon</b>	D				D		I
<b>li2MultiEllipse</b>		D			D		
<b>li2MultiEllipticalArc</b>	D	D			D		D
<b>li2MultiDot</b>						D	
<b>li2MultiPolyline - 2D</b>					D		D
<b>li2MultiPolyline - 3D</b>			D		I		D
<b>li2MultiRect</b>	D				D		I
<b>li2MultiSimplePolygon - 2D</b>	D				D		I
<b>li2MultiSimplePolygon - 3D</b>	I	D			I		I
<b>li2TriangleList</b>	D				D		I
<b>li2TriangleStrip</b>	D				D		I

## *Input Data for LI-2 and LI-3*

All input data to LI-2 or LI-3 functions are in device coordinates unless otherwise noted.

## *Picking*

It is the responsibility of the LI-1 primitive functions to handle picking, if picking is enabled. If a particular device pipeline can do picking with its hardware, then the pipeline can either cache pick hits as they occur, or it can immediately write them into the XGL core pick buffer. In the former case, if the XGL core pick buffer requires synchronization (because a software function is about to be used to pick a particular primitive, or the API has called `xgl_pick_get_identifiers()`), then the LI-1 `lilPickBufferFlush()` function is called to transfer the cached hardware pick information (if any). For example, if the device pipeline picks an object, and then the software pipeline is called to pick another object with the same pick ID, the `lilPickBufferFlush()` function is called, and only one pick is placed in the pick buffer. The XGL core merges device pipeline LI-1 pick events and LI-3 pick events.

If Z-buffering is enabled, the geometry is passed to LI-2, which will examine the current attributes and scan convert wide lines. The geometry is then passed down to LI-3 functions. LI-1 and LI-2 will have already pruned the geometric data to be inside the pick aperture; LI-3 functions must test if the geometry is visible based upon the Z comparison method. The 3D LI-3 primitive functions return a Boolean parameter *picked*. This parameter returns `TRUE` if the primitive was picked via Z-buffer-based picking (if Z-buffering is on and picking is on).

---

**Note** – LI-3 functions are only called to do picking if Z-buffering is enabled.

---



## *Hints for Rendering Transparent 3D Surfaces*

The device pipeline can optimize the rendering of transparent surfaces or let the software pipeline handle the rendering of transparent surfaces. To handle transparency, the pipeline must first determine whether the surface is transparent, and then it can decide whether to optimize the rendering of the surface. Transparency is available for 3D surfaces only. Follow these steps:

1. After face distinguishing has been done, determine whether the surface is transparent or opaque. You can use the `XgliUtilsTransparent` utilities to determine this; see Chapter 10, “Utilities” for information on these utilities.
2. Determine what action to take based on the XGL attribute `XGL_3D_CTX_BLEND_DRAW_MODE`. Optimized surface rendering will use the device’s accelerated pipeline to draw the interior of opaque surfaces and/or edges. Add the following lines of code to your LI-1 implementation of each 3D surface primitive. Add the code after front and back distinguishing and after determining whether the surface is opaque or transparent.

```
if (surface is not transparent /* returned by the util */){
    if (blend draw mode == XGL_BLEND_DRAW_NOT_BLENDED){
        // draw opaque surface but not the edges
        if (XGL_CTX_SURF_EDGE_FLAG is TRUE){
            // you may do this by setting edges to off and
            // then restoring the edge flag later
            // set things up so that only INTERIOR is drawn,
            // edges are NOT drawn
            // continue drawing the interior
        } else if (blend draw mode == XGL_BLEND_DRAW_BLENDED) {
            // draw nothing or draw edges only if edges are on
            // if (XGL_CTX_SURF_EDGE_FLAG is TRUE) {
            //     // you may do this by setting the interior to
            //     // empty and later restoring the fill style
            //     // set things up so that only EDGES are drawn,
            // } else {
            //     return 1; // nothing needs to be drawn
            // }
        }
    } else // surface is transparent {
        if (blend draw mode == XGL_BLEND_DRAW_NOT_BLENDED)
            return 1; // nothing needs to be drawn
        call the software pipeline
    }
}
```

To let the software pipeline handle all rendering of the surfaces, add the following lines of code to your implementation of each 3D surface primitive. Add the code after front and back face distinguishing and after determining whether the surface is opaque or transparent.

```
if (surface is not transparent) {
    if (blend draw mode != XGL_BLEND_DRAW_ALL) {
        call the software pipeline
    }
} else // surface is transparent {
    if (blend draw mode == XGL_BLEND_DRAW_NOT_BLENDED)
        return 1; // nothing needs to be drawn
    call the software pipeline
}
```

If a device does face distinguishing and face culling in hardware, it can optimize code that calls the software pipeline for transparency. You can use an algorithm similar to that shown in the next section on texture mapping.

## *Calling the Software Pipeline for Texture Mapping*

The application can enable texture mapping for front or back surfaces. Two conditions must be met for texture mapping to be enabled: the application must provide texture mapping objects using the RGB window raster, and the input point type must be a data point type if the texture coordinate source is DATA. If a device has not implemented texture mapping, it can call the software pipeline to do texturing.

The device pipeline can optimize its call to the software pipeline for texture mapping by determining whether the application has requested texture mapping for only front surfaces or only back surfaces. In either of those cases, the device pipeline can determine whether the surface is textured and call the software pipeline only if texture mapping is required. The following code sample provides an example.

```
if (!ctx->getSurfFaceDistinguish()){
    if (ctx->getFrontTexturing())
        // call the software pipeline
}
else {
    if (ctx->getSurfFaceCull() == XGL_CULL_BACK) {
        if (&& ctx->getFrontTexturing())
```

```

        // call the software pipeline
    }
    else if (ctx->getSurfFaceCull() == XGL_CULL_FRONT) {
        if (&& ctx->getBackTexturing())
            // call the software pipeline
        }
    else { // No culling
        if (&& (ctx->getFrontTexturing() ||
                ctx->getBackTexturing()))
            // call the software pipeline
        }
    }
}

```

In this code sample, the device pipeline first determines whether face distinguishing is enabled. If it is not, it checks whether texture mapping objects are present for texturing front faces. If so, the device pipeline calls the software pipeline. If face distinguishing is enabled, the code determines whether back faces are culled and front texture mapping is on. If so, the front surfaces can be sent to the software pipeline for texture mapping. Similarly, if front faces are culled and texture mapping is on for back surfaces, then the back surfaces can be sent to the software pipeline. Finally, if face culling is not enabled and either front or back texture mapping is enabled, the device pipeline can send all the surfaces to the software pipeline for texture mapping.

At the LI-2 level, face distinguishing has already taken place, so it is sufficient to determine whether texture mapping is enabled for front or back surfaces (based on the front flag in the *PrimData level 0* field):

```

if (ctx->get{Front,Back}Texturing())
    // fall back to the software pipeline

```

LI-1 stores the *w* component for 3D surface primitives. The *w* values are passed to LI-2 as part of the point list for 3D surface primitives.

## Antialiasing and Dithering

If your device does not perform antialiasing or dithering in hardware, your pipeline can use the software pipeline. For software pipeline dithering, check for the following attribute values and then call the software pipeline.

```
if ((device->getColorType() == XGL_COLOR_RGB) &&
    (device->getRealColorType() != XGL_COLOR_INDEX)

// if hw doesn't perform dithering, fall back to the software pipe
swp->lil{primitive}());
```

For software pipeline antialiasing, check for the value of these attributes.

For 3D markers:

```
if (ctx->getMarkerAaBlendEq() != XGL_BLEND_NONE ||
    ctx->getMarkerAaFilterWidth() > 1)
// fall back to swp if hw doesn't handle antialiasing of dots
swp->lilMultiMarker(pl);
```

For 3D strokes:

```
if (curr_stroke->getAaBlendEq() != XGL_BLEND_NONE ||
    curr_stroke->curr_stroke->getAaFilterWidth() > 1)
// fall back to swp if hw doesn't handle antialiasing of strokes
swp->lilMultiPolyline(bbox, num_lists, pl);
```

## Mapping of API Primitive Calls to LI-1 Functions

Table 9-4 shows the mapping of the 2D API primitives to the 2D LI-1 functions.

*Table 9-4* Mapping of 2D Primitives to 2D LI-1 Functions

API Function	LI-1 Functions
xgl_annotation_text()	lilAnnotationText()
xgl_context_display_gcache()	lilDisplayGcache()
xgl_multiarc()	lilMultiArc()
xgl_multicircle()	lilMultiCircle()
xgl_multimarker()	lilMultiMarker()

*Table 9-4 Mapping of 2D Primitives to 2D LI-1 Functions*

<b>API Function</b>	<b>LI-1 Functions</b>
<code>xgl_multipolyline()</code>	<code>lilMultiPolyline()</code>
<code>xgl_multirectangle()</code>	<code>lilMultiRectangle()</code>
<code>xgl_multi_simple_polygon()</code>	<code>lilMultiSimplePolygon()</code>
<code>xgl_nurbs_curve()</code>	<code>lilNurbsCurve()</code>
<code>xgl_polygon()</code>	<code>lilPolygon()</code>
<code>xgl_stroke_text()</code>	<code>lilStrokeText()</code>

Table 9-5 shows the mapping of the 3D API primitives to the 3D LI-1 functions.

*Table 9-5 Mapping of 3D API Primitives to 3D LI-1 Functions*

<b>API Function</b>	<b>LI-1 Function</b>
<code>xgl_annotation_text()</code>	<code>lilAnnotationText()</code>
<code>xgl_context_display_gcache()</code>	<code>lilDisplayGcache()</code>
<code>xgl_multiarc()</code>	<code>lilMultiArc()</code>
<code>xgl_multicircle()</code>	<code>lilMultiCircle()</code>
<code>xgl_multi_elliptical_arc()</code>	<code>lilMultiEllipticalArc()</code>
<code>xgl_multimarker()</code>	<code>lilMultiMarker()</code>
<code>xgl_multipolyline()</code>	<code>lilMultiPolyline()</code>
<code>xgl_multirectangle()</code>	<code>lilMultiRectangle()</code>
<code>xgl_multi_simple_polygon()</code>	<code>lilMultiSimplePolygon()</code>
<code>xgl_nurbs_curve()</code>	<code>lilNurbsCurve()</code>
<code>xgl_nurbs_surface()</code>	<code>lilNurbsSurf()</code>
<code>xgl_polygon()</code>	<code>lilPolygon()</code>
<code>xgl_quadrilateral_mesh()</code>	<code>lilQuadrilateralMesh()</code>
<code>xgl_stroke_text()</code>	<code>lilStrokeText()</code>
<code>xgl_triangle_list()</code>	<code>lilTriangleList()</code>
<code>xgl_triangle_strip()</code>	<code>lilTriangleStrip()</code>

Table 9-6 shows the mapping of the API raster and pixel operators to the LI-1 functions.

*Table 9-6 Mapping of API Utility Functions to LI-1 Functions*

API Function	LI-1 Function
<code>xgl_context_accumulate()</code>	<code>lilAccumulate()</code>
<code>xgl_context_clear_accumulation()</code>	<code>lilClearAccumulation()</code>
<code>xgl_context_new_frame()</code>	<code>lilNewFrame()</code>
<code>xgl_context_copy_buffer()</code>	<code>lilCopyBuffer()</code>
<code>xgl_context_flush()</code>	<code>lilFlush()</code>
<code>xgl_context_get_pick_identifiers()</code>	<code>lilPickBufferFlush()</code>
<code>xgl_context_get_pixel()</code>	<code>lilGetPixel()</code>
<code>xgl_image()</code>	<code>lilImage()</code>
<code>xgl_context_set_multi_pixel()</code>	<code>lilSetMultiPixel()</code>
<code>xgl_context_set_pixel()</code>	<code>lilSetPixel()</code>
<code>xgl_context_set_pixel_row()</code>	<code>lilSetPixelRow()</code>

## What You Should Know about the Software Pipeline

### Software Pipeline Multiplexing

The software pipeline at a particular level can call other functions for processing at the same level or at a lower level. It does this through the `opsVec` array. For example, the software pipeline LI-1 multipolyline function calls the LI-2 layer multipolyline function with the following call:

```
(dp->*((void(XglDpCtx3d:*)(XglPrimData*))
      (dp->opsVec[XGLI_LI2_MULTIPOLYLINE])
      ))(pd);
```

If the device pipeline has implemented multipolyline functionality at the LI-2 layer, it will assume control at this point; otherwise, the `ops_vector` setting will forward the rendering call directly back to the software pipeline.

Some software pipeline functions call other LI-1 functions to perform operations. For example, the software pipeline stroke text `lilStrokeText()` function calls the device pipeline LI-1 multipolyline function through the `opsVec` to display a string of text as multipolylines. As another example, the software pipeline `lilDisplayGcache()` function calls the LI-1 primitive that corresponds to the type of geometry in the Gcache. The software pipeline calls back the device pipeline through the `opsVec` array; the array pointer for that primitive determines whether the device pipeline or the software pipeline will render the primitive.

### *Software Pipeline Backing Store*

The software pipeline functions include a flag for backing store control. This flag, `do_retained`, is set by XGL core and passed to the software pipeline. However, this flag does not exist in device pipeline functions, and the device pipeline does not need to include it in calls to the software pipeline functions.

### *Surface Color in the Software Pipeline*

Surface color selection is handled as follows for software pipeline LI-1 and LI-2 functions:

- At LI-1, if lighting is on, color selection is applied before lighting. If there is no lighting, color selection is done at LI-2.
- At LI-1, if depth cueing is on and depth cue interpolation is on, then depth cueing is done at each vertex, and the depth cued color is stored at the vertices of the output point list.

If depth cue interpolation is off, and if incoming point list has vertex color (as a part of the point type or due to vertex lighting), then depth cueing is done at each vertex, and the output color is stored at the vertices of the output point list. However, if the incoming point list has facet color or if the color is obtained from the Context object, then depth cueing is done only once per facet, and the depth-cued color is stored in the first vertex of each point list.

- Color selection is performed at LI-2, if there is no lighting or depth cueing.
- If texture mapping is on, no color selection is done at LI-1, and all color selection is done at LI-2.

## *Texture Mapping in the Software Pipeline*

If texture mapping is enabled (the application must have defined at least one Data Map Texture object or Texture Map object), the software pipeline processing of surface primitives changes.

### *Texture Mapping at LI-1*

At LI-1, if texturing is enabled, the surface primitive is not lit, since the diffuse color for lighting is not known until LI-3 when texturing takes place. Therefore, lighting coefficients are computed at LI-1 and stored in the XglPrimData object. In addition, depth cueing is deferred until LI-3.

### *Texture Mapping at LI-2*

At LI-2, the following steps occur:

1. The (u,v) values for the span are computed using hyperbolic interpolation and are passed to LI-3 using the class XgliUvSpanInfo3d.
2. The lighting coefficients, if present are also computed at the spans and passed to LI-3 using XgliUvSpanInfo3d.
3. The MipMap level in which the start of the span is located and the delta is computed and passed to LI-3 using XgliUvSpanInfo3d.

See the section beginning on page 343 for information on XgliUvSpanInfo3d.

### *Texture Mapping at LI-3*

At LI-3, the (u,v) value and the lighting coefficient at a pixel are determined. The (u,v) value is used to look up the texture map to obtain the texture value (texel). Depending on the control parameters present in the Texture Map object, the texel is combined with the pixel color to obtain the final textured pixel (lighting and depth cueing are done as applicable). Note that there may be more than one texture active, and the final textured pixel is the result after applying all active textures. See page 343 for more information.

Note that the device pipeline must implement texture mapping at LI-3 or use RefDpCtx. RefDpCtx performs all operations for rendering at the LI-3 level.



---

## *LI-1 Functions*

### *About the LI-1 Layer*

The LI-1 layer specifies the loadable interface that lies just below the XGL API. An XGL loadable pipeline writer for a high-end graphics platform that supports a broad range of functionality would probably choose the LI-1 interface as the basis for an XGL port. Functions within the LI-1 layer implement the geometry pipeline for each primitive; that is, for graphics primitive operations, the functions take as an argument the points defining the primitive, and transform, light (for the 3D case), clip, and depth cue (for the 3D case) in preparation for the rendering operations performed in LI-2 and LI-3.

### *LI-1 Operations in the Software Pipeline*

Processing within the LI-1 layer transforms and clips points to device coordinates that can be used for rendering. The following operations are performed within the software pipeline LI-1 layer:

1. Model clip.
2. Transform vertices from model coordinates to world coordinates.
3. Process face culling and face distinguishing.
4. Light vertices (if necessary).
5. Transform vertices from world coordinates to device coordinates.
6. View clip. If necessary, perform rational w-clip (object is clipped to two planes  $w = \pm \epsilon$ ) and divide by  $w$ .
7. Pick the primitive.
8. Divide by  $w$ .
9. Depth cue.

Thus, an LI-1 device pipeline implementation of a primitive will incorporate these operations, although the order of the operations may differ.

## Mapping of LI-1 to LI-2 Functions in the Software Pipeline

While the LI-1 interface has a nearly one-to-one mapping of functions defined at the XGL API, at LI-2 there is a more limited set of functions. Complex LI-1 primitives, such as stroke text, do not have a corresponding interface at LI-2; instead, these primitives must be converted to a mix of simpler, component primitives before rendering.

Table 9-7 shows the mapping of the 2D LI-1 primitives to the LI-1 or LI-2 functions that they call. A hardware port of a given primitive at the LI-1 layer is free to ignore the layers below it and call whatever functions it wishes.

*Table 9-7 Mapping of 2D LI-1 Functions to LI-1 and LI-2 Functions*

LI-1 Function	LI-1 or LI-2 Function
<code>li1AnnotationText</code>	<code>li1MultiPolyline</code>
<code>li1MultiArc</code>	<code>li2MultiEllipticalArc</code> <code>li1MultiPolyline</code> <code>li1Polygon</code>
<code>li1MultiCircle</code>	<code>li2MultiEllipse</code> <code>li1Polygon</code>
<code>li1MultiMarker</code>	<code>li1MultiPolyline</code> <code>li2MultiDot</code>
<code>li1MultiPolyline</code>	<code>li2MultiPolyline</code>
<code>li1MultiRectangle</code>	<code>li1MultiSimplePolygon</code>
<code>li1MultiSimplePolygon</code>	<code>li1Polygon</code>
<code>li1NurbsCurve</code>	<code>li1MultiMarker</code> <code>li1MultiPolyline</code>
<code>li1Polygon</code>	<code>li2GeneralPolygon</code>
<code>li1StrokeText</code>	<code>li1MultiPolyline</code>

Table 9-8 shows the mapping of the 3D LI-1 primitives to LI-1 or LI-2 functions.

*Table 9-8 Mapping of 3D LI-1 Functions to LI-1 or LI-2 Functions*

<b>LI-1 Function</b>	<b>LI Function</b>
<code>lilAnnotationText</code>	<code>lilMultiPolyline</code>
<code>lilMultiArc</code> - regular arcs	<code>(open)lilMultiPolyline</code> <code>(closed)lilMultiSimplePolygon</code>
<code>lilMultiArc</code> - annotation arcs	<code>(open)li2MultiPolyline</code> <code>(closed)li2GeneralPolygon</code>
<code>lilMultiCircle</code> - regular circles	<code>lilMultiSimplePolygon</code>
<code>lilMultiCircle</code> - annotation circles	<code>li2GeneralPolygon</code>
<code>lilMultiEllipticalArc</code> - regular arcs	<code>(open)lilMultiPolyline</code> <code>(closed)lilMultiSimplePolygon</code>
<code>lilMultiEllipticalArc</code> - annotation elliptical arcs	<code>(open)li2MultiPolyline</code> <code>(closed)li2GeneralPolygon</code>
<code>lilMultiMarker</code>	<code>lilMultiPolyline</code> <code>li2MultiDot</code>
<code>lilMultiPolyline</code>	<code>li2MultiPolyline</code>
<code>lilMultiRectangle</code> - regular rectangles	<code>lilMultiSimplePolygon</code>
<code>lilMultiRectangle</code> - annotation rectangles	<code>li2GeneralPolygon</code>
<code>lilMultiSimplePolygon</code>	<code>lilPolygon</code> <code>li2MultiSimplePolygon</code>
<code>lilNurbsCurve</code>	<code>lilMultiMarker</code> <code>lilMultiPolyline</code>
<code>lilNurbsSurf</code>	<code>lilMultiMarker</code> <code>lilTriangleList</code> <code>lilQuadrilateralMesh</code> <code>lilMultiPolyline</code>
<code>lilPolygon</code>	<code>li2GeneralPolygon</code>
<code>lilQuadrilateralMesh</code>	<code>li2TriangleStrip</code>

*Table 9-8 Mapping of 3D LI-1 Functions to LI-1 or LI-2 Functions (Continued)*

<b>LI-1 Function</b>	<b>LI Function</b>
li1StrokeText	li1MultiPolyline
li1TriangleList	li2TriangleStrip li2TriangleList li2GeneralPolygon
li1TriangleStrip	li2GeneralPolygon li2TriangleStrip

### *LI-1 Attributes*

The following sections provide information on the attributes that each LI-1 primitive must handle. For more information on the attributes that a particular primitive must handle, see the man page for the primitive.

## *li1AnnotationText() - 2D/3D*

### **Overview**

`li1AnnotationText()` renders text on a plane parallel to the display surface. See the `xgl_annotation_text` man page for information on functionality that the device pipeline needs to handle.

### **Syntax**

```
[2D]void XglDpCtx2d::li1AnnotationText(  
    void*          string,  
    Xgl_pt_f2d*    ref_pos,  
    Xgl_pt_f2d*    ann_pos);  
  
[3D]void XglDpCtx3d::li1AnnotationText(  
    void*          string,  
    Xgl_pt_f3d*    ref_pos,  
    Xgl_pt_f3d*    ann_pos);
```

### **Input Parameters**

<i>string</i>	A NULL-terminated C-style list of characters if the character encoding is single-string encoding, or a pointer to a <i>Xgl_mono_text_list</i> structure if the character encoding is multi-string encoding.
<i>ref_pos</i>	The reference point for the text string.
<i>ann_pos</i>	<i>ann_pos</i> is added to the transformed reference position to obtain the annotation point.

### **Attributes That the Device Pipeline Needs to Handle**

The device pipeline must account for some or all of the attributes listed in the `xgl_annotation_text` man page.

### **Description of the Software Pipeline *li1AnnotationText* Function**

The software pipeline `li1AnnotationText()` function takes as input a single point, which is the reference position for the starting point of the string. The reference point is clipped against the window boundaries, and if it is outside the window boundaries, no text is drawn, even if part of the string would be visible.

If the reference point is inside the window boundaries, the function sets up the viewing matrices and translates the string input into points in a multipolyline point list. The multipolyline is set up to draw the text at *ann\_pos* at the appropriate scale. The function provides API arguments for input to the multipolyline interface and calls `lilMultiPolyline()`.

If the device pipeline falls back to the software pipeline to render `lilAnnotationText()`, the device pipeline `lilMultiPolyline()` function is called.

## *li1DisplayGcache() - 2D/3D*

### **Overview**

The `li1DisplayGcache()` function renders the geometry stored in the XGL device-independent Gcache object. The `li1DisplayGcache()` operator is different from other LI-1 functions because the Gcache object is an object that can contain any of a number of different primitives. See the *Solaris XGL Reference Manual* for information on Gcache functions and attributes.

The main parameter to `li1DisplayGcache()` is a pointer to an XGL Gcache object. This object is the implementation of the API Gcache functions, so it has member functions for getting and setting attributes and functions for caching incoming primitives. The pipeline does not need to use these functions.

The pipeline's task is to determine what type of primitive the Gcache contains, and then access the data for that primitive and display it. However, there is relatively little advantage for a pipeline to supply its own

`li1DisplayGcache()` function because the software pipeline handles all the work of determining the primitive, casting the data structures to the correct primitive, and calling the LI-1 function for that primitive. For example, if a text string is stored in a Gcache, the text is converted into polylines; then, when `xgl_display_gcache()` is called, the XGL core ensures that the cached polyline is rendered with the correct stroke attributes, and the software pipeline `li1DisplayGcache()` function accesses the data for the polylines and calls the `li1MultiPolyline()` function (which may or may not be supported by the device pipeline). Note that the software pipeline calls LI-1 functions for each of the Gcache primitives.

A device pipeline would want to implement `li1DisplayGcache()` for one of two reasons:

1. If for some reason the performance gain is such that the device pipeline can access the stored data faster than the XGL core can, and if the applications that the device pipeline implementor wants to capture will be using Gcaches extensively, it may be worth the effort to implement `li1DisplayGcache()`.
2. The XGL architecture provides a mechanism for the device pipeline to store device-dependent data within the Gcache. The device pipeline may be able to access the device-dependent data more efficiently than if device-

independent Gcache data is used. Alternately, the device pipeline may choose to store the data on the device and then keep a pointer to the data in the Gcache; this approach would cut down on data transport to the device.

Information on implementing `li1DisplayGcache()` and the mechanism for storing device-dependent data in the Gcache is provided below.

## Syntax

```
Xgl_cache_display XglDpCtx{2,3d}::li1DisplayGcache(
    XglGcache*    gcache
    Xgl_boolean    test,
    Xgl_boolean    display);
```

## Input Parameters

<i>gcache</i>	Pointer to an XGL Gcache object.
<i>test</i>	Boolean value that determines whether the saved state of the Gcache (attribute settings) is compared with the current state in the Context. See the <code>xgl_context_display_gcache</code> man page for more information.
<i>display</i>	Boolean value that determines whether the Gcache is rendered. See the <code>xgl_context_display_gcache</code> man page for more information.

## Attributes That the Device Pipeline Needs to Handle

A device pipeline implementation of `li1DisplayGcache()` must handle the attributes for each primitive that may be called to render the geometry in the Gcache.

## What You Need to Know to Implement `li1DisplayGcache()`

If the device pipeline supplies an implementation of the `li1DisplayGcache()` function, it will have to use the Gcache object in `XglGcache.h` as well as any one of a number of other objects that are defined as `XglGcachePrim` objects in `XglGcachePrim*.h`. The Gcache primitive objects are:



XGL_GCACHE_PRIM_MARKER	Markers
XGL_GCACHE_PRIM_MPLINE	Multipolylines
XGL_GCACHE_PRIM_TEXT	Stroke text
XGL_GCACHE_PRIM_PGON	Polygons
XGL_GCACHE_PRIM_MSPG	Multisimple polygons
XGL_GCACHE_PRIM_TSTRIP	Triangle strips
XGL_GCACHE_PRIM_NURBS_CURVE	NURBS curves
XGL_GCACHE_PRIM_NURBS_SURF	NURBS surfaces
XGL_GCACHE_PRIM_MELLA	Multi elliptical arcs

The `lilDisplayGcache()` function does the following: First, it processes the arguments *test* and *display* appropriately. Secondly, it calls the Gcache object `getOrigPrimType()` function, and then, depending on the original primitive type, the `lilDisplayGcache()` function uses the Gcache object `getGcachePrim()` function to get a pointer to the object representing the cached geometry. The pointer must be cast to the object for that primitive type.

The following code from the software pipeline `lilDisplayGcache()` function illustrates the sequence of events in rendering for each of the Gcache primitive types. You can copy or modify this source code sample as long as the resulting code is used to create a loadable pipeline for XGL.

```
XglSwpCtx3dDef::lilDisplayGcache(Xgl_gcache    gcache_obj,
                                Xgl_boolean    test,
                                Xgl_boolean    display,
                                Xgl_boolean    do_retained)
{
    XglGcache*      gcache;
    XglGcachePrim*  prim;
    Xgl_cache_display  ret_val;
    Xgl_boolean        do_display;
    Xgl_usgn32         num_model_clip_planes;

    gcache = (XglGcache*) gcache_obj;

    prim = gcache->getGcachePrim();
    if (prim == NULL) {
        return (XGL_CACHE_NOT_CHECKED);
    }

    if ((prim->getDisplayPrimType() != XGL_PRIM_NONE) &&
        !prim->getSavedCtxIs3d()) {
        return(XGL_CACHE_NOT_CHECKED); /* ctx dims don't match;
                                         best fit */
    }
}
```

```

    }

    if (test) {
        if ((prim->getDisplayPrimType() != XGL_PRIM_NONE) &&
            (prim->validate(ctx))) {
            do_display = display;
            ret_val = XGL_CACHE_DISPLAY_OK;
        }
        else {
            do_display = FALSE;
            ret_val = XGL_CACHE_ATTR_STATE_DIFFERENT;
        }
    }
    else {
        do_display = display;
        ret_val = XGL_CACHE_NOT_CHECKED;
    }
    if ((prim->getDisplayPrimType() == XGL_PRIM_NONE) ||
        !do_display)
        return ret_val;

    if (prim->wasModelClipped() &&
        ((ret_val == XGL_CACHE_DISPLAY_OK) ||
         gcache->getBypassModelClip())) {
        num_model_clip_planes = ctx->getModelClipPlaneNum();
        xgl_object_set(ctx, XGL_3D_CTX_MODEL_CLIP_PLANE_NUM, 0, 0);
    }
    else
        num_model_clip_planes = 0;

    switch (gcache->getOrigPrimType()) {
    case XGL_PRIM_STROKE_TEXT:
    {
        XglGcachePrimText*gp_text = (XglGcachePrimText *)
                                     gcache->getGcachePrim();

        Xgl_geom_status status;

        if (gp_text->getDisplayPtListList()->num_pt_lists < 1)
            return ret_val;

        xgl_context_check_bbox(ctx, XGL_PRIM_MULTIPOLYLINE,
                               gp_text->getPlm()->get_pll_bbox(), &status);
        if ((status & XGL_GEOM_STATUS_VIEW_REJECT) ||
            (status & XGL_GEOM_STATUS_MODEL_REJECT)) return ret_val;
    }
    }

```

```

        XGLI_3D_DP(void, XGLI_LI1_MULTIPOLYLINE,
                    (Xgl_bbox*, Xgl_usgn32, Xgl_pt_list*,
                     Xgl_boolean),
                    (NULL, gp_text->getDisplayPtListList()
                     ->num_pt_lists,
                     gp_text->getDisplayPtListList()->pt_lists,
                     FALSE))
    }
    break;

case XGL_PRIM_NURBS_SURFACE:
{
    XglGcachePrimNSurf*gp_nsurf = (XglGcachePrimNSurf*)
                                   gcache->getGcachePrim();

    void*   cache_data = gp_nsurf->getCacheData();
    if(cache_data == NULL){
        XglNurbsSurfData* apiData = gp_nsurf->getApiData();
        if(apiData->surface->order_u == 1 ||
           apiData->surface->order_v == 1) {
            Xgl_pt_list plist;

            plist.pt_type = apiData->surface->ctrl_pts.pt_type;
            plist.num_pts = apiData->surface->ctrl_pts.num_pts;
            plist.bbox = NULL;
            plist.pts.f3d = apiData->surface->ctrl_pts.pts.f3d;

            ctx->assignCurStrokeAsMarker();

            XGLI_3D_DP(void, XGLI_LI1_MULTIMARKER,
                        (Xgl_pt_list*, Xgl_boolean),
                        (&plist, FALSE))

            ctx->assignCurStrokeAsLine();
            break;
        }
        else {
            XglSwpNurbs nurbs(ctx, viewGrpItf, TRUE);

            cache_data = nurbs.setUsrData(gp_nsurf->getApiData(),
                                         gp_nsurf->getGcacheMode(), TRUE);
            gp_nsurf->setCacheData(cache_data);
        }
    }
}

```

```

        XGLI_3D_DP(void, XGLI_LI1_NURBS_SURFACE,
                    (Xgl_nurbs_surf*, Xgl_trim_loop_list*,
                     Xgl_nurbs_surf_simple_geom*,
                     Xgl_surf_color_spline*,
                     Xgl_surf_data_spline_list*, void*,
                     Xgl_boolean),
                    (NULL, NULL, NULL, NULL, NULL, cache_data,
                     FALSE))
    }
    break;

case XGL_PRIM_NURBS_CURVE:
{
    XglGcachePrimNCurve*gp_ncurve =
        (XglGcachePrimNCurve *)gcache->getGcachePrim();

    void*    cache_data = gp_ncurve->getCacheData();
    if(cache_data == NULL){
        XglNurbsCurveData* apiData = gp_ncurve
            ->getApiData();
        if(apiData->curve->order == 1) {
            Xgl_pt_list plist;

            plist.pt_type = apiData->curve->ctrl_pts.pt_type;
            plist.num_pts = apiData->curve->ctrl_pts.num_pts;
            plist.bbox = NULL;
            plist.pts.f3d = apiData->curve->ctrl_pts.pts.f3d;

            ctx->assignCurStrokeAsMarker();

            XGLI_3D_DP(void, XGLI_LI1_MULTIMARKER,
                        (Xgl_pt_list*, Xgl_boolean),
                        (&plist, FALSE))

            ctx->assignCurStrokeAsLine();
            break;
        }
        else {
            XglSwpNurbs nurbs(ctx, viewGrpItf, TRUE);

            cache_data = nurbs.setUsrData
                (gp_ncurve->getApiData(),
                 gp_ncurve->getGcacheMode(), TRUE);
            gp_ncurve->setCacheData(cache_data);
        }
    }
}

```

```

    }
    ctx->assignCurStrokeAsLine();

    XGLI_3D_DP(void, XGLI_LI1_NURBS_CURVE,
                (Xgl_nurbs_curve*, Xgl_bounds_fld*,
                 Xgl_curve_color_spline*, void*,
                 Xgl_boolean),
                (NULL, NULL, NULL, cache_data, FALSE))

    }
    break;

case XGL_PRIM_TRIANGLE_STRIP:
{
    XglGcachePrimTstrip*gp_tstrip =
        (XglGcachePrimTstrip*)gcache->getGcachePrim();

    register int    i;
    register Xgl_pt_list_list*display_pll =
        gp_tstrip->getDisplayPtListList();
    register Xgl_facet_list_list*display_fll =
        gp_tstrip->getDisplayFacetListList();

    for (i = 0; i < display_pll->num_pt_lists; i++) {
        XGLI_3D_DP(void, XGLI_LI1_TRIANGLE_LIST,
                    (Xgl_facet_list*, Xgl_pt_list*,
                     Xgl_tlist_flags,
                     Xgl_boolean),
                    (NULL, &(display_pll->pt_lists[i]),
                     XGL_TLIST_FLAG_USE_VTX_FLAGS, FALSE))
    }
    else
    for (i = 0; i < display_fll->num_facet_lists; i++) {
        XGLI_3D_DP(void, XGLI_LI1_TRIANGLE_STRIP,
                    (Xgl_facet_list*, Xgl_pt_list*,
                     Xgl_boolean),
                    (&(display_fll->facet_lists[i]),
                     &(display_pll->pt_lists[i]), FALSE))
    }
}
break;

case XGL_PRIM_POLYGON:
{
    Xgl_boolean do_orig_pgon;

```

```

Xgl_boolean          edges;
XglGcachePrimPgon*gp_pgon = (XglGcachePrimPgon*)
                                gcache->getGcachePrim();

do_orig_pgon = FALSE;

if ((gcache->getDisplayPrimType() ==
    XGL_PRIM_MULTI_SIMPLE_POLYGON) &&
    (gcache->getDoPolygonDecomp())) {

    /* The pgon has been decomposed */
    /* into a list of triangle stars */
    Xgl_surf_fill_stylefill_style;
    Xgl_pt_list_list*decomp_pll;
    Xgl_pt_list_list*display_pll;
    Xgl_facet_list*decomp_fl;
    Xgl_boolean      front_facing = TRUE;
    Xgl_pt_f3d* normal;
    Xgl_booleando_silhouette = FALSE;
    Xgl_boolean use_front_attributes,
                use_back_attributes;
    Xgl_surf_fill_style front_style;
    Xgl_surf_fill_style back_style;
    Xgl_boolean      distinguish;
    Xgl_surf_cull_mode cull_mode;

    decomp_pll = gp_pgon->getDecompPtListList();
    display_pll = gp_pgon->getDisplayPtListList();
    decomp_fl = gp_pgon->getDecompFacetList();

    front_style = ctx->getSurfFrontFillStyle();
    back_style = ctx->getSurfBackFillStyle();
    cull_mode = ctx->getSurfFaceCull();
    distinguish = ctx->getSurfFaceDistinguish();

    /* find out what attributes will be used */
    if (distinguish) {
        switch (cull_mode) {
            case XGL_CULL_OFF:
                use_front_attributes = TRUE;
                use_back_attributes = TRUE;
                break;

            case XGL_CULL_BACK:

```

```

        use_front_attributes = TRUE;
        use_back_attributes = FALSE;
        break;

        case XGL_CULL_FRONT:
            use_front_attributes = FALSE;
            use_back_attributes = TRUE;
            break;
        }
    }
else {
    use_front_attributes = TRUE;
    use_back_attributes = FALSE;
}

if (use_front_attributes)
    fill_style = front_style;
else
    fill_style = back_style;

/* see if orig pgon data must be used to avoid seeing */
/* the tessalation */
if (distinguish) {
    if (cull_mode == XGL_CULL_FRONT &&
        back_style == XGL_SURF_FILL_HOLLOW)
        do_orig_pgon = TRUE;
    else if (cull_mode == XGL_CULL_BACK &&
             front_style == XGL_SURF_FILL_HOLLOW)
        do_orig_pgon = TRUE;
}
else if (front_style == XGL_SURF_FILL_HOLLOW)
    do_orig_pgon = TRUE;

/* now the biggie - see if we need to determine the pgon */
/* facing */
if (!do_orig_pgon &&
    use_front_attributes && use_back_attributes &&
    front_style != back_style &&
    (front_style == XGL_SURF_FILL_HOLLOW ||
     back_style == XGL_SURF_FILL_HOLLOW)) {

    /* determine if pgon is front or back facing */
    switch (decomp_fl->facet_type) {
        case XGL_FACET_NORMAL:
            normal = &(decomp_fl->facets.normal_facets->normal);

```

```

break;

    case XGL_FACET_COLOR_NORMAL:
normal = &(decomp_fl->facets.color_normal_facets
->normal);
break;
}

/* if culled were done */
front_facing = (XgliUtFaceDistinguish(ctx, normal,
display_pll->pt_lists->pts.f3d, viewGrpItf) ==
ctx->getSurfFrontAttr3d());
if (front_facing && (cull_mode == XGL_CULL_FRONT))
return ret_val;
if (!front_facing && (cull_mode == XGL_CULL_BACK))
return ret_val ;

if (front_facing)
fill_style = front_style;
else
fill_style = back_style;
}
do_silhouette = ctx->getSurfSilhouetteEdgeFlag();

if (!gcache->getShowDecompEdges() &&
!(do_silhouette &&
fill_style == XGL_SURF_FILL_EMPTY &&
ctx->getSurfEdgeFlag() == FALSE) &&
fill_style == XGL_SURF_FILL_HOLLOW)
do_orig_pgon = TRUE;

if (!do_orig_pgon) {
if (gp_pgon->getPgonConvex()) {
XGLI_3D_DP(void, XGLI_LI1_MULTI_SIMPLE_POLYGON,
(Xgl_facet_flags, Xgl_facet_list*, Xgl_bbox*,
Xgl_usgn32, Xgl_pt_list*, Xgl_boolean),
(gp_pgon->getMspgFlags(), decomp_fl,
display_pll->bbox, 1,
display_pll->pt_lists, FALSE))
}
else {
edges = ctx->getSurfEdgeFlag();

if (edges && !gcache->getShowDecompEdges())

```



```

        ctx->setSurfEdgeFlag(FALSE);

/* pgon was decomposed into an msp list */
decomp_pll = gp_pgon->getDecompPtListList();

if(decomp_pll->num_pt_lists){
    XGLI_3D_DP(void, XGLI_LI1_MULTI_SIMPLE_POLYGON,
        (Xgl_facet_flags, Xgl_facet_list*,
        Xgl_bbox*, Xgl_usgn32, Xgl_pt_list*,
        Xgl_boolean), (gp_pgon->getMspgFlags(),
        gp_pgon->getDisplayFacetListList()
        ->facet_lists, decomp_pll->bbox,
        decomp_pll->num_pt_lists,
        decomp_pll->pt_lists, FALSE))
}
}

/* turn edges on and render orig polygon as empty */
if (edges && !gcache->getShowDecompEdges() && !gp_pgon
->getPgonConvex()) {
    ctx->setSurfEdgeFlag(TRUE);
    if (front_facing)
        ctx->setSurfFrontFillStyle(XGL_SURF_FILL_EMPTY);
    else
        ctx->setSurfBackFillStyle(XGL_SURF_FILL_EMPTY);

    Xgl_pt_list_list*pgon_pll = gp_pgon
        ->getPgonPtListList();

    if (pgon_pll->num_pt_lists == 0) {
        return ret_val;
    }

    if(pgon_pll->num_pt_lists){
        XGLI_3D_DP(void, XGLI_LI1_POLYGON,
            (Xgl_facet_type, Xgl_facet*, Xgl_bbox*,
            Xgl_usgn32, Xgl_pt_list*,
            Xgl_boolean),
            (gp_pgon->getPgonFacetType(),
            gp_pgon->getPgonFacetPtr(),
            pgon_pll->bbox,
            pgon_pll->num_pt_lists,
            pgon_pll->pt_lists, FALSE))
    }
}

```

```

/* restore fill style */
if (front_facing)
    ctx->setSurfFrontFillStyle(fill_style);
else
    ctx->setSurfBackFillStyle(fill_style);
}
}

if (gcache->getDisplayPrimType() == XGL_PRIM_POLYGON ||
    do_orig_pgon) {
if (gcache->getUseApplGeom()) {
    Xgl_pt_list_list*appl_pll = gp_pgon
        ->getApplPtListList();

    if(appl_pll->num_pt_lists == 0)
        return ret_val;

    XGLI_3D_DP(void, XGLI_LI1_POLYGON,
        (Xgl_facet_type, Xgl_facet*,
         Xgl_bbox*, Xgl_usgn32,
         Xgl_pt_list*, Xgl_boolean),
        (gp_pgon->getPgonFacetType(),
         gp_pgon->getPgonFacetPtr(),
         appl_pll->bbox,
         appl_pll->num_pt_lists,
         appl_pll->pt_lists, FALSE))
}
else {
    Xgl_pt_list_list*pgon_pll = gp_pgon
        ->getPgonPtListList();

    if (pgon_pll->num_pt_lists == 0) {
        return ret_val;
    }

    XGLI_3D_DP(void, XGLI_LI1_POLYGON,
        (Xgl_facet_type, Xgl_facet*,
         Xgl_bbox*, Xgl_usgn32,
         Xgl_pt_list*, Xgl_boolean),
        (gp_pgon->getPgonFacetType(),
         gp_pgon->getPgonFacetPtr(),
         pgon_pll->bbox,
         pgon_pll->num_pt_lists,
         pgon_pll->pt_lists, FALSE))
}
}
}

```

```

    }
    break;

    case XGL_PRIM_ELLIPTICAL_ARC:
    {
        XglGcachePrimMella*gp_mella = (XglGcachePrimMella
            *)gcache->getGcachePrim();

        if(gp_mella->getDisplayPtListList()->num_pt_lists < 1)
            return ret_val;

        if (gcache->getDisplayPrimType() ==
            XGL_PRIM_MULTIPOLYLINE) {
            XGLI_3D_DP(void, XGLI_LI1_MULTIPOLYLINE,
                (Xgl_bbox*,Xgl_usgn32,Xgl_pt_list*,
                Xgl_boolean),
                (NULL, gp_mella->getDisplayPtListList()
                ->num_pt_lists,
                gp_mella->getDisplayPtListList()
                ->pt_lists, FALSE))
        }
        else if (gcache->getDisplayPrimType() ==
            XGL_PRIM_MULTI_SIMPLE_POLYGON) {
            XGLI_3D_DP(void, XGLI_LI1_MULTI_SIMPLE_POLYGON,
                (Xgl_facet_flags, Xgl_facet_list*,
                Xgl_bbox*,
                Xgl_usgn32, Xgl_pt_list*,
                Xgl_boolean),
                (XGL_FACET_FLAG_SHAPE_CONVEX,
                gp_mella->getDisplayFacetListList()
                ->facet_lists,
                NULL,
                gp_mella->getDisplayPtListList()
                ->num_pt_lists,
                gp_mella->getDisplayPtListList()
                ->pt_lists,
                FALSE))
        }
    }
    break;

    case XGL_PRIM_MULTI_SIMPLE_POLYGON:
    {
        Xgl_boolean          do_orig_pgon;
        Xgl_boolean          edges;
    }

```

```

XglGcachePrimMspg*gp_mspg = (XglGcachePrimMspg *)
    gcache->getGcachePrim();
    Xgl_pt_list_list*      pll;
    Xgl_facet_list_list*   fll;
    Xgl_usgn32             mspg_flags;
Xgl_usgn32npl;

do_orig_pgon = FALSE;
edges = ctx->getSurfEdgeFlag();

Xgl_surf_fill_style front_style;
Xgl_surf_fill_style back_style;
Xgl_boolean         distinguish;
Xgl_surf_cull_mode  cull_mode;

front_style = ctx->getSurfFrontFillStyle();
back_style = ctx->getSurfBackFillStyle();
cull_mode = ctx->getSurfFaceCull();
distinguish = ctx->getSurfFaceDistinguish();

if (!distinguish && front_style == XGL_SURF_FILL_HOLLOW)
do_orig_pgon = TRUE;
else if (cull_mode == XGL_CULL_OFF &&
    (front_style == XGL_SURF_FILL_HOLLOW ||
    (back_style == XGL_SURF_FILL_HOLLOW && distinguish)))
do_orig_pgon = TRUE;
else if (cull_mode == XGL_CULL_FRONT &&
    (distinguish && back_style == XGL_SURF_FILL_HOLLOW))
do_orig_pgon = TRUE;
else if (cull_mode == XGL_CULL_BACK &&
    front_style == XGL_SURF_FILL_HOLLOW)
do_orig_pgon = TRUE;

if (do_orig_pgon || edges) {
if (gcache->getUseApplGeom()) {
    if(npl = gp_mspg->getApplPtListList()->num_pt_lists) {
        mspg_flags = gp_mspg->getApplMspgFlags();
        pll = gp_mspg->getApplPtListList();
        fll = gp_mspg->getApplFacetListList();
    }
}
else {
    if(npl = gp_mspg->getDisplayPtListList()->num_pt_lists) {
        mspg_flags = gp_mspg->getApplMspgFlags();

```

```

        pll = gp_mspg->getDisplayPtListList();
        fll = gp_mspg->getDisplayFacetListList();
    }
}
else {
if(npl = gp_mspg->getDisplayPtListList()->num_pt_lists) {
    mspg_flags = gp_mspg->getMspgFlags();
    pll = gp_mspg->getDisplayPtListList();
    fll = gp_mspg->getDisplayFacetListList();
}
}

if(npl) {
    XGLI_3D_DP(void, XGLI_LI1_MULTI_SIMPLE_POLYGON,
                (Xgl_facet_flags, Xgl_facet_list*,
                 Xgl_bbox*,
                 Xgl_usgn32, Xgl_pt_list*,
                 Xgl_boolean),
                (mspg_flags, fll->facet_lists, NULL, npl,
                 pll->pt_lists, FALSE))
}
}
break;

case XGL_PRIM_MULTIMARKER:
{
    XglGcachePrimMarker*gp_marker = (XglGcachePrimMarker
                                     *)gcache->getGcachePrim();
    Xgl_pt_list_list*      pll = gp_marker
                                ->getDisplayPtListList();

    if (pll->num_pt_lists < 1)
        return ret_val;

    XGLI_3D_DP(void, XGLI_LI1_MULTIMARKER,
                (Xgl_pt_list*, Xgl_boolean),
                (pll->pt_lists, FALSE))
}
break;

case XGL_PRIM_MULTIPOLYLINE:
{
    XglGcachePrimMpline*gp_mpline = (XglGcachePrimMpline
                                     *)gcache->getGcachePrim();

```

```

Xgl_pt_list_list*      pll = gp_mpline
                        ->getDisplayPtListList();

if (pll->num_pt_lists < 1)
    return ret_val;

XGLI_3D_DP(void, XGLI_LI1_MULTIPOLYLINE,
             (Xgl_bbox*, Xgl_usgn32, Xgl_pt_list*,
              Xgl_boolean),
             (NULL, pll->num_pt_lists, pll->pt_lists,
              FALSE))
}

default:
break;

} /* end switch */

if (num_model_clip_planes > 0)
    ctx->setModelClipPlaneNum(num_model_clip_planes);

return ret_val;
}

```

### ***XglGcache Functions Relevant to the Pipeline***

The device pipeline does not need to use many of the functions in `XglGcache.h`. The functions relevant to the pipeline are listed in Table 9-9.

**Table 9-9** Gcache Interfaces

Function	Description
<code>getOrigPrimType()</code>	Returns the type of the original primitive.
<code>getGcachePrim()</code>	Returns the type of the cached primitive.
<code>getPlm()</code>	Returns a point list manager object. See <code>PlManager.h</code> .
<code>getFlm()</code>	Returns a facet list manager object. See <code>FlManager.h</code> .

### ***Implementing a Device-Dependent Gcache***

The device-dependent (DD) Gcache facility allows the device pipeline to store device-dependent information with the Gcache. This device-dependent information may allow the device pipeline to render the Gcache more efficiently than the XGL core; however, the device-independent part of the Gcache remains, and it is available for the device to use. This allows the device to fall back to the software pipeline for the display of the Gcache.

The API interface to the Gcache object does not change. The application will not know about the DD part of a Gcache. The validation of a Gcache is still be done by the the XGL core; only the display of the Gcache is device dependent.

The approach for device-dependent Gcache information is that the device pipeline translates the DI Gcache primitive information into its own format. The device pipeline then associates this translation with the DI Gcache. There is a protocol between the DI Gcache and the device pipeline to manage the life cycle of the translation.

### ***Semantics of Device-Dependent Gcache***

There is no explicit function to create a DD Gcache. When the device pipeline's `lilDisplayGcache()` function is called, it can implicitly create a DD translation. It is then up to the device pipeline to associate this translation with the DI Gcache. Once the association is made, the DI Gcache and the device must tell each other to remove the DD translation for the DI Gcache. The following actions will cause the DD translation to be removed from the DI Gcache:

1. The *device* is destroyed (where device is one of DpCtx, DpDev, DpMgr, or DpLib).
2. The device decides the translation is no longer valid (e.g. out of resource, attributes changed, etc).
3. The DI Gcache is destroyed.
4. The DI Gcache gets a new primitive.

The DI Gcache and the device use a translation identifier to refer to the DD translation. This ID must be an address so it is unique within the XGL system; we suggest that it be the address of a DpCtx, DpDev, DpMgr, or DpLib object. This allows the device to choose the scope of the DD translation; that is, the

translation could be valid for just a DpCtx or for all DpCtx's associated with a DpMgr. Thus, the ID is the address of an object already under the control of the device pipeline.

The same ID is used for all Gcaches. Thus, for example, for every Gcache, the pipeline can use the address of the DpCtx object as the identifier to show that this DD translation belongs to this DpCtx object. The ID of a DD translation is the same for all the translations under the control of the device pipeline. (Remember that for each unique Context and Device pair, there is a unique DpCtx object, so if there are two Contexts associated with a Device, there is a unique DpCtx object for each Context.)

## ***XglGcache Functions for Managing a DD Translation***

In the `XglGcache.h` class, the following functions allow a device to manage a DD translation.

```
Xgl_boolean
addDdTranslation(XgliDdGcacheTranslation* dd_trans)
```

The device pipeline tells the DI Gcache to add a new translation. The new object is added to the DI Gcache's store of translations. `TRUE` is returned if the operation was successful, and `FALSE` is returned otherwise. It is up to the device pipeline to delete `dd_trans` if the addition fails.

Note that it is not guaranteed that a DI Gcache will accept a DD translation. The device must ask the Gcache object whether it will accept the translation. Therefore, code should be written to anticipate the case in which a DD translation is not accepted. In the current implementation, a Gcache can store only one DD translation. If more than one DD translation is added, the second translation is not be allowed. In the future, more than one DD translation per device/context pair may be allowed.

```
XgliDdGcacheTranslation*
lookUpTranslation(void* dd_trans_id,
                  Xgl_boolean* room_for_more)
```

The device pipeline uses this function to look up a `dd_trans_id` to see if this Gcache has a `dd_trans` for this pipeline. If the look up fails, `NULL` is returned. In this case, the `room_for_more` boolean indicates if the DI Gcache will accept new translations. If it is `FALSE`, the the pipeline should not bother building a new translation and calling `addDdTranslation()`.



```
void  
removeDdTranslation(void* dd_trans_id)
```

This function is invoked by the pipeline to tell the DI Gcache to remove the DD translation with the given ID. The XGL core does not delete the translation. This function enables the device to clean up its database of DD translations before deleting the DD translation object. This is easier to do with an explicit destroy function (which is only called by the DI Gcache) than by having the XGL core delete the translation. The DI Gcache will call the destroy function, and then the DD code should clean up the translation and delete the object.

### ***DD Translation Object***

The class `XgliDdGcacheTranslation` in the file `DdGcacheTranslation.h` acts as a wrapper for the DD translation. This class provides a standard way of handling DD translations. Devices should subclass this class for their own use. When the pipeline subclasses this class, it can then add its own data for the DD Gcache.

Thus, to create a DD translation object, the pipeline subclasses the `XgliDdGcacheTranslation` class, creates an object of the subclass, and returns a pointer to the object and casts the pointer as an `XgliDdGcacheTranslation` pointer.

### ***Example for Device-Dependent Gcache***

The following pseudo-code is an example device pipeline `lilDisplayGcache()` function. The example shows how the Gcache object functions are used to manage the DD translations. You can copy or modify this source code sample as long as the resulting code is used to create a loadable pipeline for XGL.

In this example, the call to `lookUpDdTranslation()` returns a pointer to the DD translation object, thus getting the ID for the DD translation. In addition, a pointer to a Boolean value that indicates whether there is room for a translation is returned. If `dd_trans` is `NULL` and `room_for_more` is `TRUE`, then a

DD Gcache object does not already exist, so the pipeline can build its own. When the DD Gcache is built, the function `addDdTranslation()` stores it in the Gcache, and it can then be displayed.

```
// DpCtx LilDisplayGcache
// This is where dd translations are build.
//

DpCtx::LilDisplayGcache(..., di_gcache, ...)
{
    Xgl_boolean                room_for_more;
    XglDdGcacheTranslation*    dd_tran;

    dd_tran = di_gcache->lookUpDdTranslation(tran_id,
                                              &room_for_more);

    if ((dd_tran == NULL) && room_for_more) {
        // build dd_tran
        dd_tran = build_my_dd_gcache();
        if (di_gcache->addDdTranslation(dd_tran) == FALSE) {
            delete dd_tran;
            dd_tran = NULL;
        }
    }

    if (dd_tran != NULL) {
        my_display_dd_gcache(dd_tran);
    }
}
```

It is up to the device pipeline to determine whether *dd\_trans* is valid for one context, one device, one particular DpMgr object (which would include all windows on a frame buffer), etc. This is a device-dependent decision. Most applications use one context and one device.

Whenever the device pipeline object that is managing the DD Gcache is destroyed, as part of the clean up sequence, the device pipeline must destroy the DD translations.

```
//
// Dp Object (DpCtx, DpDev, DpMgr, DpLib) which has Gcache
// translation is destroyed:

Dp::destroy_translations()
```

```
{
    // DI Gcache removes translations given translation ID
    // assuming that the pipeline has a list of device-
    // dependent translations
    // if one translation is destroyed, manage the list

    foreach dd_trans in list of translations {
        dd_trans->getDiGcache()->removeDdTranslation(trans_id);
        // clean up next/prev pointers
        delete dd_trans;
    }
}

//
// Sample XgliDdGcacheTranlation destroy function
void
XgliDdGcacheTranlation::destroy()
{
    // unlink this dd translation from lists
    // does device-dependent operations to release resources
    // and clean up data structures

    delete this;
}
```

## *li1MultiArc() - 2D*

### **Overview**

This function draws a list of arcs in the plane of the view surface. See the `xgl_multiarc()` man page for a description of the input data structure and for a description of the functionality that the device pipeline needs to handle.

### **Syntax**

```
void XglDpCtx2d::li1MultiArc(
    Xgl_arc_list *arc_list);
```

### **Input Parameters**

*arc\_list*                      A pointer to the list of arcs to draw.

### **Attributes That the Pipeline Needs to Handle**

The device pipeline must handle some or all of the attributes listed in the `xgl_multiarc()` man page.

### **Description of the Software Pipeline 2D li1MultiArc Function**

The software pipeline 2D `li1MultiArc()` function first constructs an `XglConicData2d` object called *arc\_data* using the argument *arc\_list* and then transforms the geometry to device coordinates. The behavior of a circle under transformation determines the shape of an arc in DC. A general 2D matrix can transform a circle into an ellipse with an arbitrary orientation of the major and minor axes. For a single call to this function, the MC-to-DC matrix is invariant. This matrix is the only entity that determines the eccentricity and orientation of all MC circles when they are transformed into DC. This function calculates these quantities once for a unit radius circle in MC. Circular arcs in MC become elliptical arcs in DC. For each arc, the center in MC maps to the center in DC via the MC-to-DC matrix; this is also true for the start and stop points. The radius determines the size of the elliptical arc in DC. The start and stop angles in DC (with the x-axis to the right and the y-axis down) increase in the counter-clockwise sense, and both angles are in the range  $[0, 2\pi]$ . The arc includes the locus on the ellipse from the start point to the stop point when moving counter-clockwise.

For clip checking, this function calculates the bounding box of the ellipse in DC corresponding to a unit radius circle in MC. Then, for each arc in MC, the function translates and scales the unit circle's bounding box in DC to give the bounds of the current ellipse in DC. The bounding box is reduced further by examining the start and stop points, as well as the center point if the arc fill style is sector. There are two trivial clip checking cases. If the specific DC bounding box is completely outside the view clip bounds in DC, then the function skips to the next arc. If the DC bounding box is completely inside the view clip bounds, the arc's center, major axis, minor axis, start angle, stop angle, start point, stop point, rotation angle, and edge flag are stored in a list, which is level 1 of *arc\_data*.<sup>1</sup>

If the DC bounding box is partly inside the view clip bounds, the function flushes the current list of elliptical arcs by calling the device pipeline `li2MultiEllipticalArc()` function with the current level of *arc\_data* set to level 1, which contains the list of arcs in DC. Then it decomposes the current arc into a point list with a utility, sets the current coordinate system in the view group interface to DC, and calls `li1Polygon()` or `li1MultiPolyline()` so that the polygon or polyline clipper handles the clipping depending on the whether the arc is filled or open, respectively. (An elliptical arc clipper would be desirable so that clipped elliptical arcs are converted into simpler elliptical arcs instead of polygons, but this is not implemented in the software pipelines of XGL 3.0.1.) After all arcs have been processed, the remaining buffered arcs (if any) are flushed by calling the device pipeline `li2MultiEllipticalArc()` function.

If picking is enabled, the function falls back on the polygon or polyline picker.

---

1. The calculated rotation angle is in the range  $[-\pi/2, \pi/2]$ . It's sense is right-handed. Since DC is oriented with the x-axis to the right and the y-axis down, angles increase in the clockwise sense..

## *li1MultiArc() - 3D*

### **Overview**

This function draws a list of arcs in the plane described by two direction vectors provided with the arc. See the `xgl_multiarc()` man page for a description of the input data structure and for a description of the functionality that the device pipeline needs to handle.

### **Syntax**

```
void XglDpCtx3d::li1MultiArc(
    Xgl_arc_list *arc_list);
```

### **Input Parameters**

*arc\_list*                      Pointer to the list of arcs to draw.

### **Attributes That the Device Pipeline Needs to Handle**

The device pipeline must handle some or all of the attributes listed in the `xgl_multiarc()` man page.

### **Description of the Software Pipeline 3D li1MultiArc Function**

The `li1MultiArc()` function processes regular 3D arcs and 3D annotation arcs. Processing is slightly different for annotated arcs.

If the data type of the input parameter *arc\_list* is `XGL_MULTIARC_F3D` or `XGL_MULTIARC_D3D` (the arcs are not annotated), this function evaluates the number of points necessary to tessellate each of the arcs in *arc\_list* and computes the unit circle. It then tessellates each of the arcs by projecting the arc onto the plane in MC described by the two direction vectors provided with the arc. If the multiarcs are open arcs, then the function calls `li1MultiPolyline()` to draw the tessellated arcs; otherwise, it calls `li1MultiSimplePolygon()` to draw the tessellated arcs.

If the data type of the input parameter *arc\_list* is `XGL_MULTIARC_AF3D` or `XGL_MULTIARC_AD3D` (the arcs are annotated), the function determines whether there is a bounding box provided for the entire multiarc data. If so, the function determines whether the bounding box is completely clipped by the model clipping or by the view clipping. If it is completely clipped, the function stops processing and returns control to the caller. Otherwise, the

---

function evaluates the number of points to be used to tessellate each of the arcs specified in *arc\_list* and computes the unit circle. Subsequently, the function model clips the reference points (center points) of the multiarcs and transforms the center points of the multiarcs from MC to DC. From each of the center points of the multiarcs and the corresponding radius, start angle, and stop angle, the function will tessellate the arc in DC, and view transform and clip the arc.

Finally, if picking is enabled, the function performs the picking operation for the arc. Otherwise, if the multiarcs are open arcs, the function will depth cue the open arcs and call `li2MultiPolyline()` to draw the open arcs. If the multiarcs are closed arcs, the function will depth cue the closed arcs and call `li2GeneralPolygon()` to draw the closed arcs.

## *li1MultiCircle() - 2D*

### **Overview**

This function draws a list of circles in the plane of the view surface. See the `xgl_multicircle()` man page for a description of the input data structure and a description of the functionality that the device pipeline needs to handle.

### **Syntax**

```
void XglDpCtx2d::li1MultiCircle(
    Xgl_circle_list *circle_list);
```

### **Input Parameters**

*circle\_list*                      Pointer to the list of circles to render.

### **Attributes That the Pipeline Needs to Handle**

The device pipeline must handle some or all of the attributes listed in the `xgl_multicircle()` man page.

### **Description of the Software Pipeline *li1MultiCircle* Function**

The software pipeline `li1MultiCircle()` function first constructs an `XglConicData2d` object called *circle\_data* using the argument *circle\_list* and then transforms the geometry to DC. A general 2D matrix can transform a circle into an ellipse with an arbitrary orientation of the major and minor axes. For a single call to this function, the MC-to-DC matrix is invariant. This matrix is the only entity that determines the eccentricity and orientation of all MC circles when they are transformed into DC. This function calculates these quantities once for a unit radius circle in MC. For each circle, the center in MC maps to the center in DC via the MC-to-DC matrix. The radius determines the size of the ellipse in DC.

For clip checking, this function calculates the bounding box of the ellipse in DC corresponding to a unit radius circle in MC. Then for each circle in MC, it translates and scales the unit circle's bounding box in DC to give the bounds of the current ellipse in DC. There are two trivial clip checking cases. If the specific DC bounding box is completely outside the view clip bounds in DC,



then the function skips to the next circle. If the DC bounding box is completely inside the view clip bounds, the ellipse's center, major axis, minor axis, rotation angle, and edge flags are stored in a list, which is level 1 of *circle\_data*.<sup>1</sup>

If the DC bounding box is partly inside the view clip bounds, the function flushes the current list of ellipses by calling `li2MultiEllipse()` with the current level of *circle\_data* set to level 1, which contains the list of ellipses in DC. Then it decomposes the current ellipse into a polygon with a utility, sets the current coordinate system in the view group interface to DC, and calls `li1Polygon()` so that the polygon clipper handles the clipping of the ellipse. (An ellipse clipper would be desirable so that ellipses are converted into elliptical arcs instead of polygons, but this is not implemented in the software pipelines of XGL 3.0.1.) After all ellipses have been processed, the remaining buffered ellipses (if any) are flushed by calling `li2MultiEllipse()`.

If picking is enabled, the function performs picking in MC because geometry is always a circle whereas the conic could be a rotated ellipse in DC, making picking more difficult to do quickly. The pick aperture is clipped to the view clip bounds in DC. If the aperture is at least partly inside, the corners of the clipped aperture are transformed to MC. The circle picking implementation is fast, but approximate. Some picks may be missed if the circle is smaller than the aperture and the aperture covers less than half of the circle.

---

1. The calculated rotation angle is in the range  $[-\pi/2, \pi/2]$ . It's sense is right-handed. Since DC is oriented with the x-axis to the right and the y-axis down, angles increase in the clockwise sense..

## *li1MultiCircle() - 3D*

### **Overview**

This function draws a list of circles in the plane described by two direction vectors provided with each circle. See the `xgl_multicircle()` man page for a description of the input data structure and a description of the functionality that the device pipeline needs to handle.

### **Syntax**

```
void XglDpCtx3d::li1MultiCircle(
    Xgl_circle_list *circle_list);
```

### **Input Parameters**

*circle\_list*                      Pointer to a list of circles to render.

### **Attributes That the Pipeline Needs to Handle**

The device pipeline must handle some or all of the attributes listed in the `xgl_multicircle()` man page.

### **Description of the Software Pipeline *li1MultiCircle* Function**

The `li1MultiCircle()` function processes regular 3D circles and 3D annotation circles. Processing is slightly different for annotated circles.

If the data type of the input parameter *circle\_list* is `XGL_MULTICIRCLE_F3D` or `XGL_MULTICIRCLE_D3D` (the circles are not annotated), this function evaluates the number of points to be used to tessellate each of the circles specified in *circle\_list* and computes the unit circle with the correct precision. The function then tessellates each of the circles specified in *circle\_list* by projecting the circle onto the plane in MC described by the two direction vectors provided with the circle, using the correct radius. Finally, it calls `li1MultiSimplePolygon()` to draw the tessellated circles.

If the data type of the input parameter *circle\_list* is `XGL_MULTICIRCLE_AF3D` or `XGL_MULTICIRCLE_AD3D` (the circles are annotated), the function determines whether there is a bounding box provided for the entire multicircle data. If so, the function determines whether the bounding box is completely clipped by the model clipping or by the view clipping. If it is completely clipped, the function will stop processing and return control to the caller.

---

Otherwise, the function evaluates the number of points to be used to tessellate each of the circles specified in *circle\_list* and computes the unit circle with the correct precision. It then model clips the reference points (center points) of the multicircles and transforms the center points of the multicircles from MC to DC. From each of the center points of the multicircles and the corresponding radius, the function will tessellate the circle in DC, and view transform and clip the circle. If picking is enabled, the function will perform the picking operation for the circle. Otherwise, it will depth cue the circle and call `li2GeneralPolygon()` to draw the circle.

## *li1MultiEllipticalArc() - 3D*

### **Overview**

This function draws a list of 3D elliptical arcs in the plane described by two direction vectors provided with each arc. See the `xgl_multi_elliptical_arc()` man page for a description of the input data structure and a description of the functionality that the device pipeline needs to handle.

### **Syntax**

```
void XglDpCtx3d::li1MultiEllipticalArc(
    Xgl_ell_list *ell_list);
```

### **Input Parameters**

*ell\_list*                      Pointer to a list of elliptical arcs.

### **Attributes That the Device Pipeline Needs to Handle**

The device pipeline must handle some or all of the attributes listed in the `xgl_multi_elliptical_arc()` man page.

### **Description of the Software Pipeline *li1MultiEllipticalArc* Function**

The `li1MultiEllipticalArc()` function processes regular 3D elliptical arcs and 3D annotation elliptical arcs. Processing is slightly different for annotated arcs.

If the data type of the input parameter *ell\_list* is `XGL_MULTIELLARC_F3D` or `XGL_MULTIELLARC_D3D` (the elliptical arcs are not annotated), this function evaluates the number of points to be used to tessellate each of the arcs specified in *ell\_list* and computes the unit circle with the correct precision. It then tessellates each of the arcs specified in *ell\_list* by projecting the arc onto the plane in MC described by the two direction vectors provided with the arc. If the multielliptical arcs are open arcs, the function calls `li1MultiPolyline()` to draw the tessellated arcs; otherwise, it calls `li1MultiSimplePolygon()` to draw the tessellated arcs.

If the data type of the input parameter *ell\_list* is `XGL_MULTIELLARC_AF3D` or `XGL_MULTIELLARC_AD3D` (the arcs are annotated), this function determines whether there is a bounding box provided for the entire multielliptical arc

data. If so, the function determines whether the bounding box is completely clipped by the model clipping or by the view clipping. If it is completely clipped, the function stops processing and return control to the caller. Otherwise, the function evaluates the number of points to be used to tessellate each of the arcs specified in *ell\_list* and computes the unit circle with the correct precision. The function model clips the reference points (center points) of the multielliptical arcs and transforms the center points of the multielliptical arcs from MC to DC. From each of the center points of the multielliptical arcs and the corresponding axes, start angle, and stop angle, and rotation angle, the function tessellates the arc in DC and view transforms and clips the arc.

If picking is enabled, the function performs the picking operation for the arc. Otherwise, if the multielliptical arcs are open arcs, it will depth cue the open arcs and call `li2MultiPolyline()` to draw the open arcs. If the multielliptical arcs are closed arcs, the function will depth cue the closed arcs and call `li2GeneralPolygon()` to draw the closed arcs.

## *li1MultiMarker() - 2D*

### **Overview**

This function draws a marker at each point in a list of points. See the `xgl_multimarker()` man page for a description of the input data structure and a description of the functionality that the device pipeline needs to handle.

### **Syntax**

```
void XglDpCtx2d::li1MultiMarker(
    Xgl_pt_list *point_list);
```

### **Input Parameters**

*point\_list*                      Pointer to a list of points.

### **Attributes That the Pipeline Needs to Handle**

The device pipeline must handle some or all of the attributes listed in the `xgl_multimarker()` man page.

### **Description of the Software Pipeline 2D *li1MultiMarker* Function**

The software pipeline routine takes a list of points, optionally with colors associated, transforms the points to DC, and clips them against the viewing planes.

If picking is enabled, the points are checked against the pick aperture. If a pick hit is detected, the function records the information and immediately returns; otherwise, the function returns after all the points have been checked. If picking is not enabled, markers are rendered according to the following rules:

- If the current marker type is `xgl_marker_dot`, then the list of clipped points is passed to `li2MultiDot()` for rendering.
- For all other marker types (predefined or user-defined), a point list is constructed that describes the marker as a series of strokes centered on each of the clipped points. The marker routine overwrites the current line attributes with the marker attributes and sets the MC-to-DC transform to the identity. (Note that the polyline routine need not be aware that it is being overwritten.) The point lists are passed to `li1MultiPolyline()` for rendering. After the marker function has completed processing the list, the polyline attributes and transforms are reset.

## *li1MultiMarker() - 3D*

### **Overview**

This function draws a marker at each point in a list of points. See the `xgl_multimarker()` man page for a description of the input data structure and a description of the functionality that the device pipeline needs to handle.

### **Syntax**

```
void XglSwpCtx3dDef::li1MultiMarker(  
    Xgl_point_list *point_list);
```

### **Input Parameters**

*point\_list*                      Pointer to a list of 3D points, optionally with colors (normals are ignored).

### **Attributes That the Pipeline Needs to Handle**

The device pipeline must handle some or all of the attributes listed in the `xgl_multimarker()` man page.

### **Description of the Software Pipeline 3D *li1MultiMarker* Function**

The software pipeline first clips the input list of points against the current set of model clipping planes (if no clipping planes are defined, this step is ignored). The points are then transformed to clip coordinates (CC) where they are clip-checked against the CC viewing volume. Those points that are found to be “inside” are transformed from CC to DC and are saved for the next step in the pipeline.

If picking is enabled and Z-buffering is not enabled, the points are checked against the pick volume, and the routine returns after determining whether the pick was successful.

If both picking and Z-buffering are enabled, then simply comparing the points to the pick volume is not sufficient. Points that are inside the pick volume are passed to `i2MultiDot()` to do Z comparisons to confirm a pick hit.

If picking is not enabled, then the marker points are optionally depth cued, and rendered according to the following rules.

- If the current marker type is `xgl_marker_dot`, then the list of clipped points is passed to `li2MultiDot()` for rendering.
- For all other marker types (predefined or user-defined), a point list is constructed that describes the marker as a series of strokes centered on each of the clipped points. The marker routine overwrites the current line attributes with the marker attributes and sets the MC-to-DC transform to the identity. (The polyline routine need not be aware that it is being overwritten.) These point lists are then passed to `li1MultiPolyline()` for rendering. After the marker function has completed processing the list, the attributes and transforms are reset.



## *li1MultiPolyline() - 2D*

### **Overview**

This function draws a list of unconnected polylines. See the man page `xgl_multipolyline()` for a description of the input data structures and a description of the functionality that the device pipeline needs to handle.

### **Syntax**

```
void XglDpCtx2d::li1MultiPolyline(
    Xgl_bbox      *bounding_box,
    Xgl_usgn32     num_pt_lists,
    Xgl_point_list *point_list);
```

### **Input Parameters**

<i>bounding_box</i>	Pointer to a bounding box structure that defines a bounding box for all the points in each polyline.
<i>num_pt_lists</i>	The number of point lists passed to the primitive.
<i>point_list</i>	Pointer to an array of point lists.

### **Attributes That the Pipeline Needs to Handle**

The device pipeline must handle some or all of the attributes listed in the `xgl_multipolyline()` man page.

### **Description of the Software Pipeline 2D *li1MultiPolyline* Function**

The software pipeline `li1MultiPolyline()` function transforms the input array of point lists to DC and clips them against the view planes if necessary. There may be colors and/or flags associated with the points, and the line width may be greater than 1, although these factors do not effect the geometric processing that is performed by this function. Note that line patterning is done after view clipping, so a view-clipped line may be drawn with a different pattern offset than a non-view clipped line.

If picking is enabled, then each vector is checked to see if it passes through the pick aperture. The multipolyline function either returns as soon as a pick hit is found, or returns after checking the list of vectors. If picking is not enabled, `li2MultiPolyline()` is called to render the polylines.

## *li1MultiPolyline() - 3D*

### **Overview**

This function draws a list of unconnected polylines from an input array of point lists. See the `xgl_multipolyline()` man page for a description of the input data structures and a description of the functionality that the device pipeline needs to handle.

### **Syntax**

```
void XglDpCtx2d::li1MultiPolyline(
    Xgl_bbox      *bounding_box,
    Xgl_usgn32     num_pt_lists,
    Xgl_point_list *point_list);
```

### **Input Parameters**

<i>bounding_box</i>	Pointer to a bounding box structure that defines a bounding box for all the points in each polyline.
<i>num_pt_lists</i>	The number of point lists passed to the primitive.
<i>point_list</i>	Pointer to an array of point lists.

### **Attributes That the Pipeline Needs to Handle**

The device pipeline must handle some or all of the attributes listed in the `xgl_multipolyline()` man page.

### **Description of the Software Pipeline *li1MultiPolyline* Function**

The software pipeline 3D `li1MultiPolyline()` function optionally model clips the polylines, if necessary (see `li1MultiMarker()` above), then transforms them to clip coordinates (CC) and clip-checks them against the viewing volume. The points are then transformed to DC for further processing.

If picking is enabled and Z-buffering is enabled, then the polylines are passed down to `li2MultiPolyline()`, from which Z-buffer comparisons can be made to confirm any pick hits.

---

If picking is enabled and Z-buffering is not enabled, the polylines are picked by determining whether they pass through the 3D pick volume. If any single piece of any of the polylines meets this criteria, then the entire list is deemed to have been picked, and the routine returns.

If picking is not enabled, the polylines are depth cued (if depth cueing is turned on), and `li2MultiPolyline()` is called to render them. These operations are applied regardless of the width of the line or the data that is present at the vertices (flags, colors, etc.).

Note that line patterning is done after view clipping, so a view-clipped line may be drawn with a different pattern offset than a non-view clipped line.

## *li1MultiRectangle() - 2D*

### **Overview**

This function draws a list of rectangles. See the `xgl_multirectangle()` man page for a description of the input data structure and a description of the functionality that the device pipeline needs to handle.

### **Syntax**

```
void XglDpCtx2d::li1MultiRectangle(
    Xgl_rect_list *rect_list);
```

### **Input Parameters**

*rect\_list*                      Pointer to a list of rectangles.

### **Attributes That the Pipeline Needs to Handle**

The device pipeline must handle some or all of the attributes listed in the `xgl_multirectangle()` man page.

### **Description of the Software Pipeline *li1MultiRectangle* Function**

The point information in the *Xgl\_rect\_list* structure is copied into a *Xgl\_pt\_list* structure as a list of 4-sided polygons. The function then calls `li1MultiSimplePolygon()` with the facet flags set to 4-sided and convex.

## *li1MultiRectangle() - 3D*

### **Overview**

This function draws a list of rectangles. See the `xgl_multirectangle()` man page for a description of the input data structure and a description of the functionality that the device pipeline needs to handle.

### **Syntax**

```
void XglDpCtx3d::li1MultiRectangle(  
    Xgl_rect_list    *rect_list);
```

### **Input Parameters**

*rect\_list*                      Pointer to a list of rectangles.

### **Attributes That the Pipeline Needs to Handle**

The device pipeline must handle some or all of the attributes listed in the `xgl_multirectangle()` man page.

### **Description of the Software Pipeline *li1MultiRectangle* Function**

The `li1MultiRectangle()` function processes regular 3D rectangles and 3D annotation rectangles. Processing is slightly different for annotated rectangles.

If the data type of the input parameter *rect\_list* is `XGL_MULTIRECT_F3D` or `XGL_MULTIRECT_D3D` (the rectangles are not annotated), this function projects each of the rectangles specified in *rect\_list* onto the plane in MC described by the two direction vectors provided with the rectangle and calls `li1MultiSimplePolygon()` to draw the rectangles.

If the data type of the input parameter *rect\_list* is `XGL_MULTIRECT_AF3D` or `XGL_MULTIRECT_AD3D` (the rectangles are annotated), this function first determines whether there is a bounding box provided for the entire multirectangle data. If so, the function determines whether the bounding box is completely clipped by the model clipping or by the view clipping. If it is completely clipped, the function will stop processing and return control to the caller. Otherwise, the function model clips the reference points of the multirectangles and transforms the reference points of the multirectangles from

MC to DC. From each of the reference points of the multirectangles and the corresponding corner max point, the function will build a rectangle in DC and view transform and clip the rectangle.

Finally, if picking is enabled, the function performs picking operation for the rectangle. Otherwise, it depth cues the rectangle and calls `li2GeneralPolygon()` to draw the rectangle.

## *li1MultiSimplePolygon() - 2D*

### **Overview**

This function draws a list of separate, single-bounded polygons. The polygons can be self-intersecting. See the `xgl_multi_simple_polygon()` man page for a description of the input data structures and a description of the functionality that the device pipeline needs to handle.

### **Syntax**

```
void XglDpCtx2d::li1MultiSimplePolygon(  
    Xgl_facet_flags  flags,  
    Xgl_facet_list   *facets,  
    Xgl_bbox         *bounding_box,  
    Xgl_usgn32       num_pt_lists,  
    Xgl_pt_list      *point_list);
```

### **Input Parameters**

<i>flags</i>	Structure specifying the kind of polygons being rendered.
<i>facets</i>	Pointer to a structure defining facet data for the polygons.
<i>bounding_box</i>	Pointer to a bounding box structure that defines a bounding box for all the points in the point list.
<i>num_pt_lists</i>	The number of point lists.
<i>point_list</i>	Pointer to an array of point lists for the polygons.

### **Attributes That the Pipeline Needs to Handle**

The device pipeline must handle some or all of the attributes listed in the `xgl_multi_simple_polygon()` man page.

### **Description of the Software Pipeline 2d li1MultiSimplePolygon Function**

The software pipeline 2D `li1MultiSimplePolygon()` function calls the more general `li1Polygon()` routine to process each polygon in the list of polygons. This is expected to change in future releases by optimizing the code in this routine to process the polygons.

## *li1MultiSimplePolygon() - 3D*

### **Overview**

This function draws a list of separate, single-bounded polygons. See the `xgl_multi_simple_polygon()` man page for a description of the input data structures and a description of the functionality that the device pipeline needs to handle.

### **Syntax**

```
void XglDpCtx2d::li1MultiSimplePolygon(
    Xgl_facet_flags  flags,
    Xgl_facet_list   *facets,
    Xgl_bbox         *bounding_box,
    Xgl_usgn32       num_pt_lists,
    Xgl_pt_list      *point_list);
```

### **Input Parameters**

<i>flags</i>	Structure specifying the kind of polygons being rendered.
<i>facets</i>	Pointer to a structure defining facet data for the polygons.
<i>bounding_box</i>	Pointer to a bounding box structure that defines a bounding box for all the points in the point list.
<i>num_pt_lists</i>	The number of point lists.
<i>point_list</i>	Pointer to an array of point lists for the polygons.

### **Attributes That the Pipeline Needs to Handle**

The device pipeline must handle some or all of the attributes listed in the `xgl_multi_simple_polygon()` man page.

### **Description of the Software Pipeline *li1MultiSimplePolygon* Function**

The software pipeline `li1MultiSimplePolygon()` function takes the input lists of points and transforms them to DC, clipping the polygons if necessary. Since clipping may introduce internal edges, a list of flags is introduced to indicate edges that should not be drawn later. The first step in processing the polygon is to determine whether it can be culled (this is an optional step that is only done if culling is enabled). If so, then the function immediately returns.



If the polygon passes the culling test, then it is model clipped if model clipping planes have been supplied by the user. After model clipping, the polygon is transformed and clip-checked in a manner analogous to markers and polylines. If lighting is set to `per_vertex` then the vertices are lit before clipping. This is only done for boundaries that are found to be inside the view-volume, or for boundaries that require clipping. Bounds that are outside the volume are not lit.

The final stage in polygon processing is to optionally depth cue the vertices. Once this is done, and if picking is not enabled, the polygon is passed down to `li2MultiSimplePolygon()` for rendering.

`li1MultiSimplePolygon()` does not render edges; this is the responsibility of `li2MultiSimplePolygon()`. In addition, `li1MultiSimplePolygon()` does not treat hollow or empty interior styles any differently than solid fills.

### **Notes**

Performance is optimized if the facet flags are set before calling `li1MultiSimplePolygon()` to be either 3 sided or 4 sided and convex, and there is no picking, no model clipping, and no silhouette edges, and the view clip is only done for plus w values. If one or more of these conditions are not satisfied (for example, picking is enabled), then `li1Polygon()` is called in a loop for each polygon.

## *li1NurbsCurve() - 2D*

### **Overview**

This function draws a NURBS curve of a specified order based on the list of knots in parameter space, the list of control points, and the parametric range. See the `xgl_nurbs_curve()` man page for a description of the input data structures and a description of the functionality that the device pipeline must handle.

---

**Note** – For information on how the software pipeline implements NURBS, see the list of references in Appendix C, “Accelerating NURBS Primitives”.

---

### **Syntax**

```
void XglDpCtx2d::li1NurbsCurve(
    Xgl_nurbs_curve      *curve,
    Xgl_bounds_fld       *range,
    Xgl_curve_color_spline *color_spline,
    void                 *gcache_rep);
```

### **Input Parameters**

<i>curve</i>	Pointer to a structure defining the geometry of the curve.
<i>range</i>	Pointer to a structure defining the parametric limits of the curve.
<i>color_spline</i>	Pointer to a structure defining the color distribution of the curve’s geometry. This argument is ignored in 2D.
<i>gcache_rep</i>	An optional pointer to the device-dependent Gcache representation.

### **Attributes That the Pipeline Needs to Handle**

The device pipeline must handle some or all of the attributes listed in the `xgl_nurbs_curve()` man page.

### ***Description of the Software Pipeline `lilNurbsCurve` Function***

The software pipeline `lilNurbsCurve()` function is a common entry point for both the regular NURBS curve primitive and the Gcache'd NURBS curve. For the regular NURBS primitive, the `void*` is set to `NULL`. If the Gcache contains a NURBS curve, then `lilDisplayGcache()` calls `lilNurbsCurve()` (if the order of the API geometry is greater than 1) with the `void` pointer set to the device-dependent Gcache storage, and the other three arguments are ignored. See page 229 for a description of device-dependent Gcache. A device pipeline can choose to support both the regular and the Gcache primitives in the same `lilNurbsCurve()` function, or support the regular primitive only and let the `lilDisplayGcache()` handle Gcache cases, or call the software pipeline in both cases.

If the order of the geometry data is 1, the Context current stroke is set to markers, and `lilMultiMarker()` is called with the list of control points. Otherwise, the routine takes the geometry data and tessellates it to a list of points in DC. Then, `lilMultiPolyline()` is called with the current coordinate system set to DC.

If the input is from a Gcache, a list of points is generated in MC. In the case of `COMBINED` and `STATIC` Gcaches, `lilMultiPolyline()` is called. In the case of a `DYNAMIC` Gcache, the points are generated in DC, and the push/pop coordinate system will be done like a regular 2D NURBS curve primitive.

## *li1NurbsCurve() - 3D*

### **Overview**

This function draws a NURBS curve of a specified order based on the list of knots in parameter space, the list of control points, and the parametric range. See the `xgl_nurbs_curve()` man page for a description of the input data structures and a description of the functionality that the device pipeline needs to handle.

---

**Note** – For more information on how the software pipeline implements NURBS, see the list of references in Appendix C, “Accelerating NURBS Primitives”.

---

### **Syntax**

```
void XglDpCtx2d::li1NurbsCurve(
    Xgl_nurbs_curve      *curve,
    Xgl_bounds_fld       *range,
    Xgl_curve_color_spline *color_spline,
    void                 *gcache_rep);
```

### **Input Parameters**

<i>curve</i>	Pointer to a structure defining the geometry of the curve.
<i>range</i>	Pointer to a structure defining the parametric limits of the curve.
<i>color_spline</i>	Pointer to a structure defining the color distribution of the curve’s geometry. This argument is only supported in 3D.
<i>gcache_rep</i>	An optional pointer to the device-dependent Gcache representation.

### **Attributes That the Pipeline Needs to Handle**

The device pipeline must handle some or all of the attributes listed in the `xgl_nurbs_curve()` man page.

### ***Description of the Software Pipeline `lilNurbsCurve` Function***

The software pipeline `lilNurbsCurve()` function is a common entry point for both the regular NURBS curve primitive and the Gcache'd NURBS curve. For the regular NURBS primitive, the `void*` is set to `NULL`. If the Gcache contains a NURBS curve, then `lilDisplayGcache()` calls `lilNurbsCurve()` (if the order of the API geometry is greater than 1) with the `void` pointer set to the device-dependent Gcache storage, and the other three arguments are ignored. See page 229 for a description of device-dependent Gcache. A device pipeline can choose to support both the regular and the Gcache primitives in the same `lilNurbsCurve()` function, or support the regular primitive only and let the `lilDisplayGcache()` handle Gcache cases, or call the software pipeline in both cases.

If the order of the geometry data is 1, the current stroke is set to marker, and `lilMultiMarker()` is called with the list of control points. Otherwise, the routine takes the geometry data and tessellates to a list of points in LC (Lighting Coordinates). Then `lilMultiPolyline()` is called with current coordinate system set to LC, and reset back when it returns. Vertex colors will be attached if a color spline is present.

If the input is from a Gcache, then in the case of `COMBINED` and `STATIC` Gcache, a list of points will be generated in MC. Then `lilMultiPolyline()` is called. In the case of a `DYNAMIC` Gcache, the points will be generated in LC, the push/pop of the coordinates will be done like a regular NURBS curve primitive.

## *li1NurbsSurf() - 3D*

### **Overview**

This function draws a NURBS surface of a specified order based on the list of knots in parameter space, the list of control points, and the trimming information. See the `xgl_nurbs_surface()` man page for a description of the input data structures and a description of the functionality that the device pipeline needs to handle.

---

**Note** – For more information on how the software pipeline implements NURBS, see the list of references in Appendix C, “Accelerating NURBS Primitives”.

---

### **Syntax**

```
void XglDpCtx3d::li1NurbsSurf(
    Xgl_nurbs_surf          *surface,
    Xgl_trim_loop_list      *trim_list,
    Xgl_nurbs_surf_simple_geom *hints,
    Xgl_surf_color_spline   *color_spline,
    Xgl_surf_data_spline_list *data_splines,
    void                    *gcache_rep);
```

### **Input Parameters**

<i>nurbs_surf</i>	Pointer to a structure defining the geometry of the surface.
<i>trim_list</i>	Pointer to a structure defining the trimmed portion of the surface.
<i>hints</i>	Pointer to a structure containing hints about the shape of the surface.
<i>color_spline</i>	Pointer to a structure describing the color distribution over the surface.
<i>data_splines</i>	Not currently implemented.
<i>gcache_rep</i>	An optional pointer to the device-specific Gcache representation.

### ***Attributes That the Pipeline Needs to Handle***

The device pipeline must handle some or all of the attributes listed in the `xgl_nurbs_surface()` man page.

### ***Description of the Software Pipeline `lilNurbsSurf` Function***

The software pipeline `lilNurbsSurf()` function is a common entry point for both the regular NURBS curve primitive and the Gcache'd NURBS curve. For the regular NURBS primitive, the `void*` is set to `NULL`. If the Gcache contains a NURBS surface, then `lilDisplayGcache()` calls `lilNurbsSurf()` (if the order of the API geometry is greater than 1) with the `void` pointer set to the device-dependent Gcache storage, and the other three arguments are ignored. See page 229 for a description of device-dependent Gcache. A device pipeline can choose to support both the regular and the Gcache primitives in the same `lilNurbsSurf()` function, or support the regular primitive only and let the `lilDisplayGcache()` handle Gcache cases, or call the software pipeline in both cases.

If the order of the geometry data is 1, the current stroke is set to marker, and `lilMultiMarker()` is called with the list of control points. Otherwise, the routine takes the geometry data and tessellates it to triangle lists, quadmeshes, or polylines (if an isoparametric curve is present). For the regular NURBS surface, vertices will be generated in LC (Lighting Coordinates). Then `lilTriangleList()`, `lilQuadrilateralMesh()`, or `lilMultiPolyline()` is called with current coordinate system set to LC. Vertex colors will be attached for triangle lists and quad meshes if a color spline is present.)

If the input is from a Gcache, in the case of COMBINED and STATIC Gcache, a list of points will be generated in MC, then `lilTriangleList()`, `lilQuadrilateralMesh()`, or `lilMultiPolyline()` is called. If it is a DYNAMIC Gcache, the points will be generated in LC, the push/pop of the coordinate systems will be done like a regular NURBS surface primitive.

## *li1Polygon() - 2D*

### **Overview**

This function draws a single polygon polygon that may, optionally, have several bounds. See the `xgl_polygon()` man page for a description of the input data structures and a description of the functionality that the device pipeline needs to handle.

### **Syntax**

```
void XglDpCtx2d::li1Polygon(
    Xgl_facet_type    facet_type,
    Xgl_facet         *facet,
    Xgl_bbox          *bounding_box,
    Xgl_usgn32         num_pt_lists,
    Xgl_pt_list       *point_list);
```

### **Input Parameters**

<i>facet_type</i>	Data type of the facets in the list.
<i>facet</i>	Pointer to a structure defining the facet data.
<i>bounding_box</i>	Pointer to a structure defining the bounding box around all the points in the array of point lists.
<i>num_pt_lists</i>	Number of point lists in the array of point lists.
<i>point_list</i>	Pointer to an array of point lists.

### **Attributes That the Pipeline Needs to Handle**

The device pipeline must handle some or all of the attributes listed in the `xgl_polygon()` man page.

### **Description of the Software Pipeline *li1Polygon* Function**

The software pipeline `li1Polygon()` function takes as input an array of point lists, each describing a boundary of a possibly multi-bounded polygon (in other words, “holes” are permitted). The lists of points are first transformed to DC and clipped if necessary. Since clipping may introduce internal edges, a list of flags is introduced to indicate edges that should not be drawn later.



---

If picking is enabled, then the resulting polygon is checked against the pick aperture, and the routine exits. If picking is not enabled, `li2GeneralPolygon()` is called to render the polygon. `li1Polygon()` does not render edges (edge rendering is the responsibility of `li2GeneralPolygon()`). In addition, it does not treat hollow or empty interior styles any differently than solid fills.

## *li1Polygon()* - 3D

### **Overview**

This function draws a single polygon polygon that may, optionally, have several bounds. See the `xgl_polygon()` man page for a description of the functionality that the device pipeline needs to handle

### **Syntax**

```
void XglDpCtx2d::li1Polygon(
    Xgl_facet_type  facet_type,
    Xgl_facet      *facet,
    Xgl_bbox        *bounding_box,
    Xgl_usgn32      num_pt_lists,
    Xgl_pt_list     *point_list);
```

### **Input Parameters**

<i>facet_type</i>	Data type of the facets in the list.
<i>facet</i>	Pointer to a structure defining the facet data.
<i>bounding_box</i>	Pointer to a structure defining the bounding box around all the points in the array of point lists.
<i>num_pt_lists</i>	Number of point lists in the array of point lists.
<i>point_list</i>	Pointer to an array of point lists.

### **Attributes That the Pipeline Needs to Handle**

The device pipeline must handle some or all of the attributes listed in the `xgl_polygon()` man page.

### **Description of the Software Pipeline *li1Polygon* Function**

The software pipeline 3D `li1Polygon()` routine is the general case 3D polygon renderer. The input is an array of point lists, each defining a boundary of the polygon. The first step in processing the polygon is to determine whether it can be culled (this is an optional step that is only done if culling is enabled). If so, then the function immediately returns.

---

If the polygon passes the culling test, then it is model clipped if model clipping planes have been supplied by the user. After model clipping, the polygon is transformed and clip-checked in a manner analogous to markers and polylines. If lighting is set to “per\_vertex” then the vertices are lit before clipping. This is only done for boundaries that are found to be inside the view-volume, or for boundaries that require clipping. Bounds that are outside the volume are not lit.

If picking is enabled (which means that rendering is disabled), it is done in a manner similar to markers and polylines. If Z-buffering is enabled, `li2GeneralPolygon()` is called to perform the Z comparisons necessary to verify a pick hit; otherwise, the boundaries are checked to determine whether any of them pass through the pick volume.

The final stage in polygon processing is to optionally depth cue the vertices. Once this is done, the polygon is passed down to `li2GeneralPolygon()` for rendering.

## *li1QuadrilateralMesh() - 3D*

### **Overview**

This function draws a quadrilateral mesh. See the man page `xgl_quadrilateral_mesh()` for a description on the input data structures and for information on functionality that the device pipeline needs to handle.

### **Syntax**

```
void XglDpCtx3d::li1QuadrilateralMesh(
    Xgl_usgn32      row_dim,
    Xgl_usgn32      col_dim,
    Xgl_facet_list  *facet_list,
    Xgl_point_list  *point_list);
```

### **Input Parameters**

<i>row_dim</i>	The number of rows of points defining the mesh.
<i>col_dim</i>	The number of columns of points defining the mesh.
<i>facet_list</i>	Pointer to a structure defining the facets.
<i>point_list</i>	Pointer to geometry data for the quad mesh.

### **Attributes That the Pipeline Needs to Handle**

The device pipeline must handle some or all of the attributes listed in the `xgl_quadrilateral_mesh()` man page.

### **Description of the Software Pipeline *li1QuadrilateralMesh* Function**

The software LI-1 quadrilateral mesh function breaks up the input quad mesh into triangle strips, one for each row of the original mesh. The `li1TriangleStrip()` routine is called for each triangle strip.

The points that are passed to `li1TriangleStrip()` are identical to those input to the quad mesh function with the exception that flags are introduced to mark edges that fall along the diagonals of each quad (see page 130). If the interior style is “hollow”, or if edges are enabled, then these diagonals should not be drawn. Quad mesh edges are drawn by the `li2TriangleStrip()` function. The edge pattern is restarted for every new row of the mesh.

## *li1StrokeText() - 2D/3D*

### **Overview**

`li1StrokeText()` renders characters defined as a collection of lines. See the `xgl_stroke_text()` man page for information on functionality that the device pipeline needs to handle.

### **Syntax**

```
[2D]void XglDpCtx2d::li1StrokeText(  
    void          *string,  
    Xgl_pt_f2d    *pos,  
    Xgl_pt_f3d    *dir);  
  
[3D]void XglDpCtx3d::li1StrokeText(  
    void          *string,  
    Xgl_pt_f3d    *pos,  
    Xgl_pt_f3d    *dir);
```

### **Input Parameters**

<i>string</i>	A NULL-terminated C-style list of characters if the character encoding is single-string encoding, or a pointer to a <i>Xgl_mono_text_list</i> structure if the character encoding is multi-string encoding.
<i>pos</i>	The reference point for the position of the string and the origin of the text plane.
<i>dir</i>	An array containing the two direction vectors used for the orientation of the 2D plane on which the text sits. Used for 3D Contexts only.

### **Attributes That the Pipeline Needs to Handle**

The device pipeline must account for some or all of the attributes listed in the `xgl_stroke_text()` man page.

***Description of the Software Pipeline `li1StrokeText` Function***

The software pipeline `li1StrokeText()` function takes as input a single point, which is the starting position for the string. This point is clipped against the window boundaries, and if it is outside the window boundaries, no text is drawn, even if part of the string would be visible.

If the point is inside the window boundaries, this function takes the string and converts it to multipolylines. It then sets the current stroke to text, sets up the viewing matrices, and calls `li1MultiPolyline()`.

## *li1TriangleList() - 3D*

### **Overview**

This function draws a triangle list, which is a set of points that form a triangle strip, a triangle star, or a group of unconnected triangles. See the `xgl_triangle_list()` man page for a description of the input data structures and for information on functionality that the device pipeline needs to handle.

### **Syntax**

```
void XglDpCtx3d::li1TriangleList(  
    Xgl_facet_list    *facet_list,  
    Xgl_pt_list       *point_list,  
    Xgl_tlist_flags   flags);
```

### **Input Parameters**

<i>facet_list</i>	Pointer to a structure defining the facet information for the triangle list.
<i>point_list</i>	Pointer to the point list defining the vertices of the triangles in the triangle list.
<i>flags</i>	A word containing information on the overall characteristics of the triangle list.

### **Attributes That the Pipeline Needs to Handle**

The device pipeline must handle some or all of the attributes listed in the `xgl_triangle_list()` man page.

### **Description of the Software Pipeline *li1TriangleList* Function**

The software pipeline function provides general purpose triangle rendering. It is more flexible than the `li1TriangleStrip()` primitive because it allows for the rendering of triangle stars and independent triangles in addition to triangle strips. However, the operations performed by this LI-1 call are similar to the `li1TriangleStrip()` function.

The first step is to branch to one of four different internal routines based on the value of the global triangle list flags parameter (part of the API call). This parameter specifies whether the input point list describes a triangle strip, a triangle list, a set of independent triangles, or a set of triangles that is composed of a combination of strips, stars and independent triangles. The points in the point list are interpreted differently based on whether the triangle list defines a triangle star, strip, or set of independent triangles. The triangle list function always tries to keep together runs of points in a point list that define one particular type of triangle; thus, a set of points defining a set of independent triangles, for instance, will be handled as one point list.

Within each of these four routines, the processing steps are similar. The first step is to model clip the triangles if model clipping is requested. Model clipping is done before culling because it may cause the triangle list to be separated into smaller sub-strips which can be processed more easily as separate entities.

Next, face culling is performed on the triangle list (if enabled by the user). This involves traversing the triangle list and skipping over those facets that are culled. This effectively breaks the triangle list down into smaller sub-lists of non-culled faces. Each of these sub-lists is then processed separately as is done for model-clipping. (The difference is that the normal-direction information calculated by this stage can be reused later on in the pipeline.) There are separate routines for culling the different kinds of triangle lists.

When the original triangle list has been broken down into sublists that are visible in MC, the vertices are transformed to clip coordinates (CC) and clip-checked in the same manner as the other 3D primitives described earlier. If lighting is enabled, the vertices that are found to be inside the clip volume, or those that require clipping, are lit in MC, and the new color is stored with the transformed points. Finally, the transformed points are depth cued, if depth cueing is enabled.

Once processing is complete on a sublist, it is ready to either be picked, or rendered. If picking is enabled and Z-buffering is enabled, `li2TriangleStrip()` or `li2TriangleList()` is called for Z-buffered picking. For non-Z-buffered picking, a simple geometry test is performed on the points to see if they lie within the pick aperture. If picking is not enabled, the points are passed down to either `li2TriangleStrip()` or `li2TriangleList()` for rendering.



---

Edges are drawn on triangle lists by calling the `li2MultiPolyline()` function for each triangle separately. This means that the edge pattern is restarted for each new triangle.

---

**Note** – If a triangle within the list requires clipping, then `li2GeneralPolygon()` is called to render it. This is because `li2TriangleStrip()` and `li2TriangleList()` require points that are in strip or list format. A clipped triangle may not necessarily still be in this format.

---

## *li1TriangleStrip() - 3D*

### **Overview**

This function draws a triangle strip. See the `xgl_triangle_strip()` man page for a description of the input data structures and for information on functionality that the device pipeline needs to handle.

### **Syntax**

```
void XglDpCtx3d::li1TriangleStrip(
    Xgl_facet_list *facet_list,
    Xgl_pt_list *point_list);
```

### **Input Parameters**

<i>facet_list</i>	Pointer to a structure containing facet information for the triangle strip.
<i>point_list</i>	Pointer to a structure defining the point list.

### **Attributes That the Pipeline Needs to Handle**

The device pipeline must handle some or all of the attributes listed in the `xgl_triangle_strip()` man page.

### **Description of the Software Pipeline *li1TriangleStrip* Function**

This function operates like `li1Polygon()`, except for the added complexity of multiple facets being derived from a one point list. The input to this routine is a single point list defining the vertices of the triangles in the strip. A facet list that defines facet normals and/or colors for the triangles can also be passed to this function.

The first step in processing the triangle strip is to optionally model clip it. This is done first, rather than face culling, because the model clipping utility may separate the strip into smaller sub-strips, and it is easier to process each of the sub-strips separately than as a group.

Next, face culling is performed on the strip (if enabled by the user). This involves traversing the strip and skipping over those facets that are determined to be culled. This effectively breaks the strip down into smaller sub-strips of non-culled faces. Each of these sub-strips is then processed

separately. (The difference between this and the separate processing of model clipping is that the normal-direction information calculated by this stage can be reused later in the pipeline.)

When the original strip has been broken down into sub-strips that are visible in MC, the vertices are transformed to clip coordinates (CC) and clip-checked in the same manner as other 3D primitives. As with `li1Polygon()`, if lighting is enabled, the vertices that are found to be inside the clip volume, or those that require clipping, are lit in MC, and the new color is stored with the transformed points. Finally, the transformed points are depth cued, if depth cueing is enabled.

Once processing is complete on a sub-strip, it is ready to either be picked, or rendered. If picking is enabled, then either `li2TriangleStrip()` is called for Z buffered picking, or a simple geometry test is performed on the points to see if they lie within the aperture. If picking is not enabled, the points are passed to `li2TriangleStrip()` for rendering.

Edges are drawn on triangle strip by calling the `li2MultiPolyline()` function for each triangle separately. This means that the pattern is restarted for each new triangle.

---

**Note** – If a triangle within the strip requires clipping, then `li2GeneralPolygon()` is called to render it. This is because `li2TriangleStrip()` requires points that are in “strip format”. A clipped triangle may not necessarily still be in this format.

---

## *li1Accumulate()* - 3D

### **Overview**

This function accumulates from one buffer to another. See the `xgl_context_accumulate()` man page for information on functionality that the device pipeline needs to handle.

### **Syntax**

```
void XglDpCtx3d::li1Accumulate(
    Xgl_bounds_i2d* rectangle,
    Xgl_pt_i2d*      position,
    float            src_wt,
    float            accum_wt,
    Xgl_buffer_set   copy_buf);
```

### **Input Parameters**

<i>rectangle</i>	Source area of the draw buffer of the raster. If <i>rectangle</i> is NULL, the maximum area of the source buffer is assigned to the value by the XGL core
<i>position</i>	The position in the destination buffer to be used as the starting position. If <i>position</i> is NULL, the top left corner of the destination buffer is assigned to the value by the XGL core.
<i>src_wt</i>	The weight to be used as the source weight in the accumulation calculation.
<i>accum_wt</i>	The weight to be used as the accumulation weight in the accumulation calculation.
<i>copy_buf</i>	The buffer to copy the accumulated image to.

---

**Note** – Note that although the application can specify NULL values for *rectangle* and *postion*, the XGL core assigns valid values to these parameters before passing them to the device pipeline; thus, the pipeline does not have to test for this but can assume the values for these parameters are valid.

---

### ***Attributes That the Pipeline Needs to Handle***

The device pipeline must handle the attributes listed in the `xgl_context_accumulate()` man page.

### ***What You Need to Know to Implement `li1Accumulate`***

Accumulation buffers must be either 32- or 48-bits. Indexed colors are not supported. The accumulation buffer must be BBGRR or XBGR.

### ***Description of the Software Pipeline `li1Accumulate` Function***

The software pipeline `li1Accumulate()` function accumulates the rectangle (`rect`) from the draw buffer to the accumulation buffer starting at `pos`. If `copy_buf` is not `XGL_BUFFER_SEL_NONE`, the accumulated area is also copied to `rect` in `copy_buf`. If `rect` is `NULL`, it is assumed to be the entire raster. If `pos` is `NULL`, it is assumed to be (0,0). If either `rect` or `pos` are out of bounds, they are both clipped accordingly.

If the weight for a buffer is zero, that buffer is not read. If the source weight is 0.0 and the destination (accumulation) weight is 1.0, the accumulation and copy-back steps are omitted (the copy to `copy_buf` is still performed, however). There is no substantial optimization at this time if the source weight is 1.0 and the destination weight is 0.0.

## *li1ClearAccumulation() - 3D*

### **Overview**

This function clears the accumulation buffer. See the `xgl_context_clear_accumulation()` man page for information on functionality that the device pipeline needs to handle.

### **Syntax**

```
void XglDpCtx3d::li1ClearAccumulation(
    Xgl_color*    color);
```

### **Input Parameters**

*color*                      Color value.

### **Attributes that the Pipeline Needs to Handle**

The device pipeline must handle the attribute listed in the `xgl_context_clear_accumulation()` man page.

### **What You Need to Know to Implement *li1ClearAccumulation***

Accumulation buffers must be either 32- or 48-bits. Indexed colors are not supported. Format XBGR pixels are accessed as words. BBGRRR pixels are normally accessed as arrays of three *Xgl\_usgn16* structures, but because the SPARC architecture is big-endian, the BBGRRR pixels are stored a word at a time as well. This software implementation is for the SPARC architecture only. The implementation will be changed to run on both the x86 architecture and the SPARC architecture for the 10/93 release.

### **Description of the Software Pipeline *li1ClearAccumulation* Function**

The software pipeline `li1ClearAccumulation()` function sets the entire buffer specified by `XGL_3D_CTX_ACCUM_OP_DEST` to the specified color. The accumulation buffer is either BBGRRR or XBGR.

## *li1CopyBuffer() - 2D/3D*

### **Overview**

This function copies a block of pixels from one buffer to another. See the `xgl_context_copy_buffer()` man page for information on functionality that the device pipeline needs to handle.

### **Syntax**

```
[2D and 3D]
void XglDpCtx{2,3}d::li1CopyBuffer(
    Xgl_bounds_i2d* rectangle,
    Xgl_pt_i2d*      position,
    XglRaster*       source_ras);
```

### **Input Parameters**

<i>rectangle</i>	Area that is copied in the source buffer. If <i>rectangle</i> is <code>NULL</code> , the maximum area of the source buffer is assigned to the value by the XGL core. The source rectangle cannot have negative components; that is, <code>xmin</code> , <code>xmax</code> , <code>ymin</code> , and <code>ymax</code> cannot be less than zero, and <code>xmin</code> and <code>ymin</code> cannot be greater than <code>xmax</code> and <code>ymax</code> respectively.
<i>position</i>	Position in the destination buffer where the copy begins. If <i>position</i> is <code>NULL</code> , the top left corner is assigned to the value by the XGL core.
<i>source_ras</i>	The buffer to be used as the source for the copy.

---

**Note** – Note that although the application can specify `NULL` values for *rectangle* and *postion*, the XGL core assigns valid values to these parameters before passing them to the device pipeline; thus, the pipeline does not have to test for this but can assume the values for these parameters are valid.

---

### **Attributes That the Pipeline Needs to Handle**

The device pipeline must handle some or all of the attributes listed in the `xgl_context_copy_buffer()` man page.

### ***What You Need to Know to Implement `lilCopyBuffer`***

Copy buffer copies a block of pixels from a buffer in system memory to the frame buffer or from the frame buffer to system memory. The direction of the copy (i.e. memory to frame buffer or vice versa) is reflected in the XGL core as follows:

- If the copy is from a memory raster to the frame buffer, `lilCopyBuffer()` is used for the copy operation. In this case, a memory raster is the source buffer, and the device associated with the Context in the `xgl_context_copy_buffer()` call is a window raster device.
- If the copy is from the frame buffer to a memory raster, the source buffer is a window raster device, and the device associated with the Context in the `xgl_context_copy_buffer()` call is a memory raster. In this case, the `XglDpDev::copyBuffer()` function is called to do the copy operation. Note that when copying from device to memory, the device object must perform the copy between `winLock()` and `winUnLock()` calls.

The XGL core determines what type of device the application is requesting for the source raster and the destination raster, and then calls the appropriate copy buffer routine.

For the case of copying from memory to a window raster, if the pipeline chooses to implement `lilCopyBuffer()`, it must take into account different color models and different underlying representations of memory. The memory raster can be indexed or RGB color type. However, XGL makes a distinction between the real color type, which is the actual memory organization for the data in the device, and the color type of the XGL Device that the application works with. For copying from memory to a window raster, the `lilCopyBuffer()` function must take into account all the cases of the various combinations of Device color type and real color type, although the pipeline may want to optimize some cases, such as the straight-forward copy from an indexed memory raster to an indexed device.

Since the `XglDpDev` copy buffer function is device-dependent and since the software pipeline does not currently implement `lilCopyBuffer()`, both copy buffer functions must be implemented by the device pipeline. However in `CopyBuffer.h`, XGL provides utility functions that perform copy operations with all the color conversion and fill styles.



`CopyBuffer.h` defines the data structures and interfaces for the copy buffer utility functions. It provides two utility functions: `XgliUtCopyBuffer()` and `XgliUtFbToMemCopyBuffer()`. `XgliUtCopyBuffer()` is a general routine that copies from one buffer to another; it can be used for either the memory to frame buffer copy or the frame buffer to memory copy.

`XgliUtFbToMemCopyBuffer()` is a wrapper on `XgliUtCopyBuffer()` that is easier to use for the frame buffer to memory copy. These copy buffer utilities use the `PixRect` object to represent the raster memory for the copy. See Chapter 4, “Internal Data Storage” for information on `PixRects`.

`XgliUtCopyBuffer()` has pointers to the following structures, which can be `NULL` if the pipeline doesn’t need them:

- Destination foreground color *dest\_fg\_color* and background color *dest\_bg\_color* can be `NULL` if the fill style is such that they are not needed.
- Destination clip mask *dest\_clip\_mask* and source clip mask *src\_clip\_mask* can be `NULL` if there is no per-pixel clip mask or if the whole area of window or memory is visible so that the pipeline does not need to do a per-pixel clip.
- The copy buffer color information structure *color\_info* needs to be filled out. This structure specifies the color space of the data. Color Map objects are needed to do the copy and color conversion.
- The *rop\_info* structure can be `NULL` if mask and rop mode do not apply or the device does not want plane mask or raster operations to be done in software.
- If the raster fill style is `XGL_RAS_FILL_COPY` (see the reference page for `XGL_CTX_RASTER_FILL_STYLE`), the *fill\_info* structure must be filled in with the fill style; otherwise, the *fill\_info* structure can be `NULL`. If there is a raster fill pattern or a stipple raster, then the `PixRect` needs to be supplied, and the stipple position and color must be supplied.
- Note that the *z\_buffer\_info* structure is not required for copy buffer, but if this utility is used for `xgl_image()`, this structure would need to be filled in.

The `RefDpCtx` utility also provides an `lilCopyBufferMemToFB()` function that the pipeline can use to implement the memory to frame buffer case of copy buffer. Note that none of the XGL-provided utilities for copying buffers are optimized, so it may be advisable for the pipeline to implement at least the more straight-forward copy operations.

In summary, here’s what you need to do to implement copy buffer:

1. Implement `li1CopyBuffer()`. You can use the `RefDpCtx` utility `li1CopyBufferMemToFB` to do this.
2. Implement `XglDpDev::CopyBuffer()` using `XgliUtFbToMemCopyBuffer()`.

---

**Note** – You must also implement the LI-3 versions of copying to and from buffers, but you can use the `RefDpCtx` utilities `li3CopyToDpBuffer()` and `li3CopyFromDpBuffer()`. See page 329 through page 331 for more information on LI-3 copy buffer functions.

---

### ***Description of the Software Pipeline li1CopyBuffer Function***

The software pipeline does not implement this function.

## *li1Flush() - 2D/3D*

### **Overview**

This function causes pending or asynchronous processing to complete. See the `xgl_context_flush()` man page for information on functionality that the device pipeline needs to handle.

### **Syntax**

```
void XglDpCtx{2,3}d::li1Flush(  
    Xgl_usgn32    flush_action);
```

### **Input Parameters**

*flush\_action*      The type of flushing that the function performs. See the man page for the options.

### **Attributes that the Pipeline Needs to Handle**

The device pipeline must handle the attributes listed in the `xgl_context_flush()` man page.

### **Description of the Software Pipeline *li1Flush* Function**

The software pipeline does not implement this function.

## *li1GetPixel() - 2D/3D*

### **Overview**

This function gets the color value of a specified pixel. See the `xgl_context_get_pixel()` man page for information on functionality that the device pipeline needs to handle.

### **Syntax**

```
[2D and 3D]
void XglDpCtx{2,3}d::li1GetPixel(
    Xgl_pt_i2d*   position,
    Xgl_color*    value,
    Xgl_boolean*  obscured);
```

### **Input Parameters**

<i>position</i>	Location of the pixel.
<i>value</i>	Location where the retrieved color value is stored.
<i>obscured</i>	TRUE if the window is covered at that pixel position.

### **Attributes That the Pipeline Needs to Handle**

The device pipeline must handle some or all of the attributes listed in the `xgl_context_get_pixel()` man page.

### **Description of the Software Pipeline *li1GetPixel* Function**

This function is not implemented by the software pipeline.

---

## *li1Image()* - 2D/3D

### **Overview**

This function displays a block of pixels from a raster. See the `xgl_image()` man page for information on functionality that the device pipeline needs to handle.

### **Syntax**

```
[2D] void XglDpCtx::li1Image(  
    Xgl_pt_f2d*    position,  
    Xgl_bounds_i2d* image,  
    XglRaster*     src_ras);  
  
[3D] void XglDpCtx::li1Image(  
    Xgl_pt_f3d*    position,  
    Xgl_bounds_i2d* image,  
    XglRaster*     src_ras);
```

### **Input Parameters**

<i>position</i>	The position in the destination Context where the copy starts. The position must be a valid point in the Context's model space.
<i>image</i>	The rectangular area in the source raster to be copied. If <i>image</i> is NULL, the maximum area of the source Raster is assigned to the value by the XGL core
<i>src_ras</i>	The source memory raster.

---

**Note** – Note that although the application can specify a NULL value for *image*, the XGL core assigns a valid value to this parameter before passing it to the device pipeline; thus, the pipeline does not have to test for this but can assume the value is valid.

---

### **Attributes That the Pipeline Needs to Handle**

The device pipeline must handle some or all of the attributes listed in the `xgl_image()` man page.

***Description of the Software Pipeline `li1Image()` Function***

The software pipeline `li1Image()` function first verifies that rectangle is within the source raster's boundaries. For 3D images, the function then does model clipping if necessary. Then, for both 2D and 3D image copying, the function transforms the position point from model coordinates to device coordinates and verifies that the position in DC is not clipped; if the position point is clipped, the image is not copied. The function then clips the rectangle against the `src_ras` boundaries and the DC bounds, verifies that the render buffer is the draw buffer, sets up the copy information, and calls `li3CopyToDpBuffer()` to do the copying.

## *li1NewFrame() - 2D/3D*

### **Overview**

This function clears the device coordinate viewport and possibly the Z-buffer. See the `xgl_context_new_frame()` man page for information on functionality that the device pipeline needs to handle.

### **Syntax**

```
[2D and 3D]
void XglDpCtx{2,3}d::li1NewFrame();
```

### **Input Parameters**

None

### **Attributes That the Pipeline Needs to Handle**

The device pipeline must handle some or all of the attributes listed in the `xgl_context_new_frame()` man page.

### **What the You Need to Know to Implement li1NewFrame**

In the case of indexed color, the plane mask during a new frame operation is different from the plane mask used for rendering. The new frame plane mask prepares the surface based on a pixel mapping offset. To simplify the processing of the new frame plane mask, the XGL core provides the inline function `getNewFramePlaneMask()`. This function can be called regardless of the color type of the device. The following example shows the use of `getNewFramePlaneMask()`.

```
if (action & XGL_CTX_NEW_FRAME_CLEAR) {
    Xgl_booleanchange_flag = FALSE;

    Xgl_usgn32new_frame_plane_mask;
    new_frame_plane_mask = baseCtx->getNewFramePlaneMask();
    if (cached_plane_mask != new_frame_plane_mask) {
        change_flag = TRUE;
        //set the new frame plane mask
    }

    // Perform the clear operation
    if (change_flag) {
```

```
        // Restore the original plane mask
    }
}
```

### ***Description of the Software Pipeline `li1NewFrame` Function***

The software pipeline does not implement this function.



## *li1PickBufferFlush() - 2D/3D*

### **Overview**

This function requires a device pipeline to empty its device pick buffer, if any, into the XGL core pick buffer. This is useful for asynchronous devices that buffer pick events. The function is called when the API function `xgl_get_pick_identifiers` is called. It also is called to synchronize the device's pick buffer and the XGL core pick buffer before each call to the software picking code. See the `xgl_get_pick_identifiers()` man page for information on functionality.

### **Syntax**

```
void XglDpCtx{2,3}d::li1PickBufferFlush();
```

### **Input Parameters**

None

### **Attributes that the Pipeline Needs to Handle**

None

### **What You Need to Know to Implement *li1PickBufferFlush***

The purpose of this function is to allow synchronization between a device's pick buffer and the pick buffer maintained by XGL's device-independent code. The device-independent picking routines call this function whenever the software pipeline detects a pick (if a pipeline has fallen back to the software pipeline to pick a particular primitive, for instance) and when the application explicitly requests to see the contents of the pick buffer (via `xgl_pick_get_identifiers()`).

To implement this function, device pipelines check the hardware pick buffer (if applicable) and then add the identifiers of the pick events to the XGL core pick buffer using the DI function `ctx->addPickToBuffer(Xgl_usgn32 pick_id1, Xgl_usgn32 pick_id2)`. If a device does not support picking, then this function need not be implemented.

The Context class includes another function that compares the last recorded pick IDs with the current pick IDs and returns `TRUE` if they are identical. This function is `checkLastPick()`. This function is an optimization to allows the

device pipeline to return to the application immediately if nothing new has been picked. Note that for devices caching pick events `checkLastPick()` does not call `lilPickBuffer Flush()`. This means that the last recorded pick event might not be the last actual pick event if the pipeline's cached pick events have not been flushed in the XGL core pick buffer.

`lilPickBufferFlush()` takes no arguments and is only called by the software pipeline and the XGL core. A device pipeline need not call this function itself.

### ***Description of the Software Pipeline `lilPickBufferFlush` Function***

The software pipeline does not implement this function.

## *li1SetMultiPixel()*

### **Overview**

This function sets the color values for a list of pixel locations. See the `xgl_context_set_multi_pixel()` man page for information on the functionality that the device pipeline needs to handle.

### **Syntax**

```
[2D and 3D]
void XglDpCtx{2,3}d::li1SetMultiPixel(
    Xgl_usgn32    count,
    Xgl_pt_i2d    *pt,
    Xgl_color     *color);
```

### **Input Parameters**

<i>count</i>	Number of pixels to write.
<i>pt</i>	Array of screen locations to write to.
<i>color</i>	Array of pixel colors to write (in one-to-one correspondence with the <i>location</i> array).

### **Attributes That the Pipeline Needs to Handle**

The device pipeline must handle some or all of the attributes listed in the `xgl_context_set_multi_pixel()` man page.

### **Description of the Software Pipeline *li1SetMultiPixel* Function**

The software pipeline `li1SetMultiPixel()` function writes a set of *count* pixels into the locations specified by the *pt* array argument. The first pixel is written to (pt[0].x, pt[0].y), the next pixel is written to (pt[1].x, pt[1].y), etc. There should be at least *count* valid entries in the *pt* argument. Since this routine operates on individual pixels, rather than geometry, all coordinates are device coordinates, and the 2D and 3D versions of this routine are identical.

Each pixel color is determined by taking successive values from the *color* argument, which should contain an array of colors, one color for each pixel. The `color[0]` entry specifies the color for the pixel located at (pt[0].x, pt[0].y), `color[1]` for (pt[1].x, pt[1].y), etc. The *count* argument specifies the number of pixels to write. The *color* argument array should have at least *count* entries.

## *li1SetPixel() - 2D/3D*

### **Overview**

This function sets the color value for a specified pixel. See the `xgl_context_set_pixel()` man page for information on functionality that the device pipeline needs to handle.

### **Syntax**

```
[2D and 3D]
void XglDpCtx{2,3}d::li1SetPixel(
    Xgl_pt_i2d    *position,
    Xgl_color     *color);
```

### **Input Parameters**

<i>position</i>	Location of the pixel value to be set.
<i>color</i>	The color value of the pixel that is set.

### **Attributes That the Pipeline Needs to Handle**

The device pipeline must handle some or all of the attributes listed in the `xgl_context_set_pixel()` man page.

### **Description of the Software Pipeline *li1SetPixel* Function**

The software pipeline does not implement this function.

## *li1SetPixelRow() - 2D/3D*

### **Overview**

This function sets the color value for a row of pixels. See the `xgl_context_set_pixel_row()` man page for information on functionality.

### **Syntax**

```
[2D and 3D]
void XglDpCtx{2,3}d::li1SetPixelRow(
    Xgl_usgn32    start_col,
    Xgl_usgn32    row,
    Xgl_usgn32    count,
    Xgl_color     *color);
```

### **Input**

<i>start_col</i>	First x-coordinate of pixel row.
<i>row</i>	y-coordinate of all pixels in pixel row.
<i>count</i>	Number of pixels to write.
<i>color</i>	Array of pixel colors to write.

### **Attributes That the Pipeline Needs to Handle**

The device pipeline must handle some or all of the attributes listed in the `xgl_context_set_pixel_row()` man page.

### **Description of the Software Pipeline *li1SetPixelRow* Function**

The software pipeline `li1SetPixelRow()` function writes a series of *count* contiguous, horizontal pixels along the y-position supplied by *row*, starting at the x-position *start\_col* and continuing in the direction of increasing x values. Since this routine operates on individual pixels, rather than geometry, all coordinates are device coordinates, and the 2D and 3D versions of the routine are identical. The pixel colors along the row are determined by taking successive values from the *color* argument, which should contain an array of colors, one color corresponding to each pixel in the row. The `color[0]` entry specifies the color for the left-most pixel located at `(start_col,row)`, `color[1]` for `(start_col + 1, row)`, etc. The *count* argument specifies the number of pixels to write. The *color* argument array should have at least *count* entries.

## *LI2 Functions*

### *About the LI-2 Layer*

Conceptually, the LI-2 layer lies below the transformation and clipping of the LI-1 layer. While the LI-1 functions implement and make decisions regarding the geometry pipeline for each primitive, the LI-2 functions are more concerned with rendering. The LI-2 layer was designed to provide support for hardware that might not be able to perform transformations and clipping but that can still accelerate DC primitives. The LI-2 layer provides a porting layer that is simpler to port to than LI-1 but that renders faster than the dot/span layer LI-3.

In general, the LI-2 renderers are passed an internal data structure containing a list of points or a list of point lists in device coordinates. These points have already been view clipped, and, in the case where the canvas is completely exposed (window rasters only), the points have been window clipped as well. The calling function is responsible for setting the attributes in the Context to be used by the LI-2 routine: for example, when rendering a hollow polygon using the polyline renderer, the line color attribute should be set to reflect the polygon color.

For 3D Contexts, the LI-2 renderers are similar to the 2D case. The calling LI-1 function should perform any applicable lighting and depth cueing; however, if depth cueing is enabled, 3D LI-2 functions must handle DC offset and interpolate colors.

### *LI-2 Surface Attributes*

Table 9-10 lists the attributes that must be accounted for by the LI-2 surface primitives. Note that `Context.h` and `Context3d.h` provide interfaces for the pipeline to get a number of 3D surface attributes within a single structure. These functions can facilitate device pipeline manipulation of 3D surface attributes. At LI-2, face determination has already taken place. Using these interfaces, a pipeline can set up the surface attribute pointer based on the

facing in the renderer and do all the attribute processing without referring to the actual facing. See page 127 and page 129 for information on these interfaces.

*Table 9-10* Surface Attributes at LI-2

	Attributes
2D and 3D	<code>getSurfFrontColor()</code> <code>getSurfFrontColorSelector()</code> <code>getSurfFrontFpat()</code> <code>getSurfFrontFpatPosition()</code> <code>getSurfFrontFillStyle()</code>  <code>getEdgeAltColor()</code> <code>getEdgeCap()</code> <code>getEdgeColor()</code> <code>getEdgeJoin()</code> <code>getEdgeMiterLimit()</code> <code>getEdgePattern()</code> <code>getEdgeStyle()</code> <code>getEdgeWidthScaleFactor()</code>  <code>getSurfEdgeFlag()</code> <code>getSurfInteriorRule()</code>  <code>getRop()</code>
3D only	<code>getSurfBackColor()</code> <code>getSurfBackColorSelector()</code> <code>getSurfBackFillStyle()</code> <code>getSurfBackFpat()</code> <code>getSurfBackFpatPosition()</code>  <code>getHlhsrMode()</code>  <code>getSurfDcOffset()</code>  <code>getDepthCueMode()</code> <code>getDepthCueInterp()</code>

## Mapping of LI-2 Functions to LI-3 Functions in the Software Pipeline

Table 9-11 shows the mapping of the 2D LI-2 functions to LI-3 functions or other LI-2 functions.

*Table 9-11 Mapping of 2D LI-2 Functions to LI-2 or LI-3 Functions*

LI-2 Function	LI-2 Function
li2GeneralPolygon()	li3MultiSpan() li2MultiPolyline() (hollow polygons or edges on)
li2MultiDot()	li3MultiDot()
li2MultiEllipse()	li3MultiSpan() li2GeneralPolygon() (rotation angles)
li2MultiEllipticalArc()	li3Vector() (straight lines of sectors or chords) li3MultiSpan() li2MultiPolyline() (rotation angles - open) li2GeneralPolygon() (rotation angles - closed)
li2MultiPolyline()	li3Vector() (thin lines) li3MultiSpan() (wide lines)
li2MultiRect()	li3MultiSpan() li2MultiPolyline() (hollow rects or edges on)
li2MultiSimplePolygon()	li3MultiSpan() li2MultiPolyline() (hollow polygons or edges on)

Table 9-12 shows the mapping of the 2D LI-2 functions to LI-3 functions and other LI-2 functions.

*Table 9-12 Mapping of 3D LI-2 Functions to LI-2 and LI-3 Functions*

LI-2 Function	LI-2 Function
li2GeneralPolygon()	li3MultiSpan() li2MultiPolyline() (hollow polygons or edges on)
li2MultiDot()	li3MultiDot()
li2MultiPolyline()	li3Vector() (thin lines) li2TriangleList() (wide lines)



Table 9-12 Mapping of 3D LI-2 Functions to LI-2 and LI-3 Functions

LI-2 Function	LI-2 Function
li2MultiSimplePolygon()	li2GeneralPolygon()
li2TriangleList()	li3MultiSpan() (filled surfaces) li2MultiPolyline() (hollow triangles or edges on)
li2TriangleStrip()	li3MultiSpan() (filled surfaces) li2MultiPolyline() (hollow triangles or edges on)

## *li2GeneralPolygon() - 2D/3D*

### **Overview**

This function scan converts a polygon to span lines. A general polygon routine supports geometry that cannot be easily tessellated (such as multi-bounded polygons) and provides an opportunity for hardware to handle such cases. The `li2GeneralPolygon()` function is expected to support different interior styles and fill rules (even-odd only), and to handle edges.

### **Syntax**

```
void XglDpCtx{2,3}d::li2GeneralPolygon(
    XglPrimData* pd);
```

### **Input Parameters**

*pd*                      Pointer to an `XglPrimData` object containing a list of point lists specifying a single (possibly multi-bounded) polygon.

### **Attributes That the Device Pipeline Needs to Handle**

See Table 9-10 on page 309 for a list of attributes that this function must handle.

### **Description of the Software Pipeline *li2GeneralPolygon***

To render filled surfaces, `li2GeneralPolygon()` scan converts the polygon to a list of span lines and calls the device pipeline `li3MultiSpan()` function to draw the list of spans. For 3D polygons, the function handles texture mapping, adds the surface DC offsets to the Z value, and calls the pipeline `li3MultiSpan()` function.

To render hollow surfaces or edges, the function converts the point list into point lists for multipolylines, sets the current stroke to hollow or edge, and calls the device pipeline `li2MultiPolyline()` function.

## *li2MultiDot() - 2D/3D*

### **Overview**

A multi-dot routine at the LI-2 level provides the opportunity for hardware to accelerate dot markers.

### **Syntax**

```
[2D and 3D]
void XglDpCtx{2,3}d::li2MultiDot(
    XglPrimData* pd);
```

### **Input Parameters**

*pd*                      Pointer to an XglPrimData object containing a list of marker positions in device coordinates.

### **Attributes That the Device Pipeline Needs to Handle**

A device pipeline must handle the following attributes.

```
XglContext::getMarkerColorSelector()
XglContext::getMarkerColor()
```

### **Description of the Software Pipeline *li2MultiDot* Function**

The software pipeline *li2MultiDot()* function determines the marker color based on the marker color selector, the input point type, or in the 3D case, the depth cueing mode. It then calls the device pipeline *li3MultiDot()* function to draw the markers.

## *li2MultiEllipse() - 2D*

### **Overview**

This function scan converts ellipses to span lines. Although there is no ellipse in the 2D API, XGL provides `li2MultiEllipse()` to support hardware that can accelerate a regular circle or a circle with uneven scale in DC. This function is expected to handle different interior fill styles and edges.

### **Syntax**

```
void XglDpCtx2d::li2MultiEllipse(  
    XglConicData2d*ellipses);
```

### **Input Parameter**

<i>ellipses</i>	Pointer to an <code>XglConicData2d</code> object containing a list of ellipses, with each ellipse specified with a center point, and a major and minor axis in DC.
-----------------	--

### **Attributes**

See Table 9-10 on page 309 for a list of attributes that this function must handle.

### **Description of the Software Pipeline *li2MultiEllipse* Function**

For ellipses without rotation angles, the software pipeline converts each ellipse to a list of span lines and calls `li3MultiSpan()` to draw the spans.

For ellipses with rotation angles, the ellipse is tessellated to a list of points, and the pipeline `li2GeneralPolygon()` is called to draw the geometry.

## *li2MultiEllipticalArc() - 2D*

### **Overview**

This function scan converts elliptical arcs to span lines. Although there is no ellipse in the 2D API, XGL provides `li2MultiEllipticalArc()` to support hardware that can accelerate a circular arc with uneven scale in DC. You may want to implement `li2MultiEllipticalArc()` if your hardware can accelerate arcs or elliptical arcs. The function is expected to handle different interior fill styles and edges, and different arc fill styles.

### **Syntax**

```
void XglDpCtx2d::li2MultiEllipticalArc(  
    XglConicData2d*arcs);
```

### **Input Parameters**

<i>arcs</i>	Pointer to an <code>XglConicData2d</code> object containing a list of partial ellipses, with each ellipse specified with a center point, major and minor axes, and a start and stop angle in DC.
-------------	--

### **Attributes**

A device pipeline must handle the following attribute in addition to the surface attributes listed in Table 9-10 on page 309.

```
XglContext::getArcFillStyle()
```

### **Description of the Software Pipeline `li2MultiEllipticalArc` Function**

For elliptical arcs without rotation angles, the software pipeline scan converts the interior and arc borders to a list of span lines and calls `i3MultiSpan()` to draw the spans. If the arcs have a fill style (`XGL_CTX_ARC_FILL_STYLE`) of `XGL_ARC_SECTOR` or `XGL_ARC_CHORD` and have thin lines for the line segments, the device pipeline LI-3 function `li3Vector()` is called to draw the lines. The line segments of arcs with a fill style of `XGL_ARC_SECTOR` or `XGL_ARC_CHORD` and with thick lines for the line segments are drawn with `li3MultiSpan()`. For ellipses with rotation angles, if the arc fill style is open, the arc is tessellated to a list of points, and `li2MultiPolyline()` is called. If the arc is closed, `li2GeneralPolygon()` is called.

## *li2MultiPolyline() - 2D*

### **Overview**

This function is expected to handle wide lines and wide patterned lines as well as thin lines and thin patterned lines.

The `li2MultiPolyline` routine is the most multiplexed of the LI-2 functions. Since this routine is called by the other stroke primitives, the polyline attributes (color, style, width, etc.) are read from the current stroke group in the Context object. It is the responsibility of the calling routine to set the stroke group appropriately for the original primitive; it is the responsibility of the LI-2 multipolyline function to get the polyline attributes from the current stroke group. See page 106 for information on the stroke group.

### **Syntax**

```
void XglDpCtx::li2MultiPolyline(
    XglPrimData* pd);
```

### **Input Parameters**

*pd*                      Pointer to an `XglPrimData` object containing point lists describing multiple, disjoint polylines. A flag that specifies whether each polyline is closed is also included.

### **Attributes**

A device pipeline must handle the following attributes.

```
XglContext::getRop()
XglContext::getCurrentStroke()
XglStrokeGroup::getAaBlendEq()
XglStrokeGroup::getAaFilterWidth()
XglStrokeGroup::getAaFilterShape()
XglStrokeGroup::getAltColor()
XglStrokeGroup::getCap()
XglStrokeGroup::getColor()
XglStrokeGroup::getColorSelector()
XglStrokeGroup::getJoin()
XglStrokeGroup::getMiterLimit()
XglStrokeGroup::getPattern()
XglStrokeGroup::getStyle()
```

```
XglStrokeGroup::getWidthScaleFactor()  
XglStrokeGroup::getExpectedFlagValue()  
XglStrokeGroup::getFlagMask()
```

### ***What You Should Know to Implement `li2MultiPolyline`***

There is a flag in `li3Vector()` that determines whether the last pixel of a line segment is drawn. To prevent drawing the shared pixel twice for consecutive lines, set the *draw\_last\_pixel* flag for `li3Vector()` to `FALSE`. Then, set it to `TRUE` for the last segment in the polyline.

In addition, when the point type has flag information, the device pipeline must check the stroke group flag mask and expected flag value to determine whether individual segments of the line should be drawn. For more information, see “Flag Mask and Expected Flag Value” on page 111.

### ***Description of the Software Pipeline 2D `li2MultiPolyline` Function***

For thin lines and thin patterned lines, the software pipeline 2D `li2MultiPolyline()` function calls `li3SetVectorControl` to set the attributes for the LI-3 line renderers and calls `li3Vector()` function to draw the lines.

To render wide lines as well as caps and joins, the software implementation creates a list of span lines. The span lines are sorted, and clipped if necessary, for certain ROP modes so that correct rendering will occur with span lines overlap. The list of spans is drawn with the `li3MultiSpan()` function.

## *li2MultiPolyline() - 3D*

### **Overview**

This function is expected to handle wide lines and wide patterned lines as well as thin lines and thin patterned lines. Device pipelines that call `li2MultiPolyline()` for wide lines may also want to implement `li2TriangleList()` for triangle stars if the device can accelerate triangles.

The `li2MultiPolyline` routine is the most multiplexed of the LI-2 functions. Since this routine is called by the other stroke primitives, the polyline attributes (color, style, width, etc.) are read from the current stroke group in the Context object. It is the responsibility of the calling routine to set the stroke group appropriately for the original primitive; it is the responsibility of the LI-2 function to get the stroke attributes from the current stroke group. See page 106 for information on the stroke group.

### **Syntax**

```
void XglDpCtx3d::li2MultiPolyline(
    XglPrimData *pd);
```

### **Input Parameters**

*pd*                      Pointer to an `XglPrimData` object containing point lists of vertex coordinates of the lines in device coordinates.

### **Attributes**

A device pipeline must handle the following attributes.

```
XglContext::getRop()
XglContext3d::getHlhrMode()
XglContext::getCurrentStroke()
XglStrokeGroup::getAaBlendEq()
XglStrokeGroup::getAaFilterWidth()
XglStrokeGroup::getAaFilterShape()
XglStrokeGroup::getAltColor()
XglStrokeGroup::getCap()
XglStrokeGroup::getColor()
XglStrokeGroup::getColorSelector()
XglStrokeGroup::getJoin()
XglStrokeGroup::getMiterLimit()
```



```
XglStrokeGroup::getPattern()  
XglStrokeGroup::getStyle()  
XglStrokeGroup::getWidthScaleFactor()  
XglStrokeGroup::getExpectedFlagValue()  
XglStrokeGroup::getFlagMask()  
XglStrokeGroup3d::getDcOffset()  
XglStrokeGroup3d::getColorInterp()
```

### ***Description of the Software Pipeline li2MultiPolyline Function***

The software pipeline 3D `li2MultiPolyline()` function breaks lines into line segments and sends the segments individually to either `li3Vector()` or `li2TriangleList()`.

If the value of the width scale factor attribute for the line is less than 2.0, the function creates individual line segments, puts each line segment into an *Xgli\_vector\_3d* structure, and then calls `li3Vector()` for rendering.

Wide lines and relevant caps and joins are converted to triangle stars. Specifically, if the line width scale factor is equal to or greater than 2.0, the function creates rectangular line segments, converts each segment into triangle stars, and calls `li2TriangleList()` for rendering. Patterned wide lines are broken down at each pattern boundary, so only solid triangle stars are sent to `li2TriangleList()`.

## *li2MultiRect() - 2D*

### **Overview**

This function scan converts rectangles to span lines and is expected to handle different interior fill styles and edges. It is provided for hardware that can accelerate rectangles and provides an opportunity to reduce the amount of data copied (2 corners versus 4 points if a polygon routine is used.)

### **Syntax**

```
void XglDpCtx2d::li2MultiRect(
    XglRectData2d* rects);
```

### **Input Parameters**

<i>rects</i>	Pointer to an XglRectData2d object containing a list of rectangles specified by their corners.
--------------	--

### **Attributes**

See Table 9-10 on page 309 for a list of attributes that this function must handle.

### **Description of the Software Pipeline *li2MultiRect* Function**

If the interior style of the surface is solid, stippled, opaque-stippled, or patterned, the software pipeline `li2MultiRect()` function scan converts the polygon to a list of span lines and calls `li3MultiSpan()` to draw the spans. If the interior style of the surface is hollow or if the edge flag is on, `li2MultiPolyline()` is called with the stroke group set to hollow or edge accordingly.

This function is not currently used by any LI-1 functions in the software pipeline. In the future, `li2MultiRect()` may be called from the `li1MultiRectangle()` function if the MC-to-DC transform does not contain any rotations.

## *li2MultiSimplePolygon() - 2D*

### **Overview**

This function scan converts polygons to span lines. It is expected to handle different fill styles and edges. This function is provided for hardware that can accelerate single-bounded polygons.

### **Syntax**

```
void XglDpCtx2d::li2MultiSimplePolygon(  
    XglPrimData* pd);
```

### **Input Parameters**

<i>pd</i>	Pointer to an XglPrimData object containing a list of point lists in device coordinates, each describing a single, bounded polygon.
-----------	---

### **Attributes**

See Table 9-10 on page 309 for a list of attributes that this function must handle.

### **Description of the Software Pipeline *li2MultiSimplePolygon* Function**

To render filled surfaces, the software pipeline scan converts the polygon to a list of span lines and calls `li3MultiSpan()` to draw the spans.

To render hollow surfaces or edges, the software pipeline sets the stroke group to hollow or edge and calls `li2MultiPolyline()`.

This function is not currently used by any LI-1 functions in the software pipeline. In the future, `li2MultiSimplePolygon()` may be called from `li1MultiSimplePolygon()`.

## *li2MultiSimplePolygon() - 3D*

### **Overview**

This function scan converts polygons to span lines and provides support for single-bounded polygons. This function is expected to handle different fill styles and edges.

### **Syntax**

```
void XglDpCtx3d::li2MultiSimplePolygon(
    XglPrimData*pd);
```

### **Input Parameters**

*pd*                      Pointer to an XglPrimData object containing a list of polygons.

### **Attributes**

See Table 9-10 on page 309 for a list of attributes that this function must handle.

### **Description of the Software Pipeline *li2MultiSimplePolygon* Function**

The software pipeline `li2MultiSimplePolygon()` function calls `li2GeneralPolygon()` in a loop for each simple polygon. In future releases, the software pipeline may provide optimized code to process the polygons.

## *li2TriangleList() - 3D*

### **Overview**

This function renders single-facing, non-mixed triangle lists in the form of triangle strips, triangle stars, or unconnected triangles in device coordinates.

### **Syntax**

```
void XglDpCtx3d::li2TriangleList(  
    XglPrimData*pd);
```

### **Input Parameters**

<i>pd</i>	Pointer to an XglPrimData object containing point lists of either single-facing triangle strips, triangle stars, or unconnected triangles based on the value of the triangle list render flags. The triangle point list cannot be mixed. All surfaces must be single facing.
-----------	--

### **Attributes**

A device pipeline must handle the following attribute in addition to the surface attributes listed in Table 9-10 on page 309.

```
XglLevel::getRenderFlags()  
XglContext3d::getDepthCueInterp()  
XglContext3d::getDepthCueMode()  
XglContext3d::getHlhrMode()
```

### **Description of the Software Pipeline *li2TriangleList* Function**

To render triangle strips, the software pipeline function checks the level data rendering flags and calls `li2TriangleStrip()`. To render filled surfaces, the function scan converts triangle stars and independent triangles into lists of spans. It handles color selection and texture mapping, adds the corresponding surface DC offsets to the Z value, and calls `li3MultiSpan()`. See page 216 for information on texture mapping. To render hollow triangles or edges, the function converts the triangle point list into point lists for multipolylines, assigns the current stroke, and calls the device pipeline `li2MultiPolyline()` function.

## *li2TriangleStrip() - 3D*

### **Overview**

This function handles single-facing triangle strips.

### **Syntax**

```
void XglDpCtx3d::li2TriangleStrip(
    XglPrimData*pd);
```

### **Input Parameters**

*pd* An XglPrimData object containing point lists of single-facing triangle strips in device coordinates.

### **Attributes**

A device pipeline must handle the following attributes in addition to the surface attributes listed in Table 9-10 on page 309.

```
XglLevel::getRenderFlags()
XglContext3d::getDepthCueInterp()
XglContext3d::getDepthCueMode()
XglContext3d::getHlhrMode()
```

### **Description of the Software Pipeline *li2TriangleStrip* Function**

The software pipeline *li2TriangleStrip()* function first processes the input point lists into separate triangles. Then, for filled surfaces, the function scan converts the triangle strips into lists of spans. It handles color selection and texture mapping, adds the corresponding DC offsets to the Z value, and calls *li3MultiSpan()*. See page 216 for information on texture mapping.

To render hollow surfaces or edges, the function converts the triangle point list into lists of points for multipolyline, handles color selection, assigns the current stroke, and calls *li2MultiPolyline()*.

## LI-3 Functions

### About the LI-3 Layer

The LI-3 layer is the lowest level of the XGL interface hierarchy. The LI-3 layer consists of control functions, primitive functions, and begin/end batching functions. It also includes functions that copy pixel data to and from buffers managed by the device pipeline. The primitive functions are listed in Table 9-13.

Table 9-13 LI-3 Primitive Functions

Function	Description
<code>li3MultiDot()</code>	Draws a list of dots.
<code>li3MultiSpan()</code>	Draws a list of horizontal spans.
<code>li3Vector()</code>	Draws a single vector.

The control functions set the attributes for LI-3 primitives. The begin/end batching functions are used to indicate that a series of the same LI-3 primitives will be sent and that the state will not change between successive calls. This allows the device pipeline some opportunities for optimization when implementing LI-3.

There is no software pipeline implementation of the LI-3 functions; therefore, each device pipeline must implement all of the LI-3 operators for 2D and 3D primitives. However, to help you with this task, there is a set of utilities called RefDpCtx (Reference Device Pipeline Context), which implements the LI-3 layer. RefDpCtx is built on a simple get-pixel and put-pixel interface; it is not meant to provide fast performance but to enable the device pipeline to get XGL running relatively quickly. See “RefDpCtx” on page 348 for more details on using RefDpCtx to implement the LI-3 layer.

Note that although the 2D versions of the primitive functions are straightforward, the 3D versions are more complicated because they must support antialiasing, shading, and texture mapping. The 3D version of LI-3 is perhaps more complex than it should be; however, you can use RefDpCtx to implement the difficult cases.

## Notes on Implementing LI-3 Functions

The main caller of LI-3 functions is the software pipeline LI-2 layer (the device pipeline could call its own LI-3 functions, but that is not very likely). The calling sequence from the software pipeline LI-2 layer to the device pipeline LI-3 layer is:

```
set Context attributes (if needed)
li3Set<Prim>Control (if needed)
li3Begin(<Prim>)
    li3<Prim> (called as many times as needed)
li3End(<Prim>)
```

where *<Prim>* is one of the LI-3 primitive functions. All device pipeline LI-3 primitives called by the software pipeline are surrounded by `li3Begin()` and `li3End()` calls. Within the `li3Begin()/li3End()` pair, only the primitive type specified in the `li3Begin()` function can be called, and no other Context or primitive functions can be called. Within a begin/end, neither the Context nor the LI-3 state will change, and the device pipeline can continue to render. `li3Begin()` returns a Boolean value: `TRUE` means that the LI-3 primitive will be visible when rendered; `FALSE` means that the LI-3 primitive function will not draw anything because the window is obscured, so the device pipeline may not want to call the primitive function.

The LI-3 implementation must take into account the color type of the Device and the color type specified by the XGL API. To do this, the LI-3 implementation may want to get the following information from the Device.

```
XglRaster::getDoPixelMapping()
XglDevice::getColorType()
XglDevice::getRealColorType()
XglDevice::getCmap()
XglDevice::getDrawable()
```

The LI-3 implementation also must be aware of the rendering buffer as specified by:

```
XglContext::getRenderBuffer()
```



All LI-3 2D functions receive geometry data in 2D integer Device Coordinates. The geometry will be within the bounds of the Device, but it is up to the LI-3 implementation to clip the primitives to the window clip list. All 3D LI-3 geometric coordinates are specified in floating 3D Device coordinates and, as such, may have fractional components for the coordinate values.

Attributes relevant to the LI-3 functions are listed on the page for each function description.

---

**Note** – As in the other LI layers, LI-3 requires that window locking be done around each LI-3 rendering call.

---

### *Picking at LI-3*

The 3D LI-3 primitive functions return a Boolean parameter *picked*. This parameter returns `TRUE` if the primitive was picked via Z-buffer-based picking (if Z-buffering is on and picking is on). LI-1 and LI-2 will have already pruned the geometric data to be inside the pick aperture; LI-3 functions must test if the geometry is visible based upon the Z comparison method.

The *picked* return value is an optimization for LI-2. If the return value is `TRUE`, then LI-2 can stop sending primitives. The software pipeline LI-2 function which calls LI-3 will update the pick buffer. It is allowable, however, for LI-3 to always return `FALSE`, but in this case, the LI-3 function must update the pick buffer by using the XglContext function `ctx->addPickToBuffer(Xgl_usgn32 pick_id1, Xgl_usgn32 pick_id2)`. The device pipeline code need only fill in the *picked* parameter if picking is enabled. If picking is disabled, it can be ignored.

---

**Note** – LI-3 functions are only called to do picking if Z-buffering is enabled.

---

### *Where to Look for More Information*



For more information on LI-3 data structures and functions, refer to the header files `Li3Structs.h`, `Li3Structs2d.h`, and `Li3Structs3d.h`. If you are planning on using `RefDpCtx`, refer to `RefDpCtx.h`, `RefDpCtx2d.h`, and `RefDpCtx3d.h`.

## *li3Begin() and li3End() - 2D/3D*

### **Syntax**

```
[2D]
Xgl_boolean XglDpCtx2d::li3Begin(
    Xgli_layer_prim_2d  prim_type);

void XglDpCtx2d::li3End(
    Xgli_layer_prim_2d  prim_type);

[3D]
Xgl_boolean XglDpCtx3d::li3Begin(
    Xgli_layer_prim_3d  prim_type);

void XglDpCtx::li3End(
    Xgli_layer_prim_3d  prim_type);
```

### **Input Parameters**

*prim\_type*                      The type of primitive that is called between the LI-3 Begin/End calls.

### **Attributes That the Device Pipeline Needs to Handle**

There are no specific attributes used by these functions.

### **What You Need to Know to Implement li3Begin and li3End**

`li3Begin()` specifies the beginning of a sequence of LI-3 primitives of type *prim\_type*; `li3End()` indicates the end of the sequence. In between the Begin/End pair, only the specified LI-3 primitive function will be called, there will not be any calls to other LI-3 functions or calls to the Context. It is permissible for the implementation of `li3Begin()` to call `WIN_LOCK` and to hold the lock until `li3End()` is called. However, the implementation must be sure that the lock does not time out; that is, the implementation may have to release and then reacquire the lock before `li3End()` is called.

`li3Begin()` returns `TRUE` if the primitive will be visible when rendered and `FALSE` if it will not be. For example, the primitive would not be visible if the window were completely covered.

## *li3CopyFromDpBuffer() - 2D/3D*

### **Syntax**

```
[2D and 3D]
void XglDpCtx2d::li3CopyFromDpBuffer(
    const Xgl_bounds_i2d*   src_rect,
    const Xgl_pt_i2d*       dest_pos,
    Xgl_buffer_sel          sel,
    XglPixRectMem*          buf);
```

### **Input Parameters**

<i>src_rect</i>	A rectangle in the device pipeline's coordinates (relative to the origin of the window).
<i>dest_pos</i>	The position to copy to (relative to the origin of <i>buf</i> ).
<i>sel</i>	Selects an image buffer in the pipeline, when the pipeline is multi-buffering, to copy out of.
<i>buf</i>	The PixRect buffer where the data is copied into.

### **Attributes That the Device Pipeline Needs to Handle**

There are no specific attributes used by this function.

### **What You Need to Know to Implement *li3CopyFromDpBuffer***

This function copies pixel data from the device pipeline's frame buffer into memory. The memory is represent by a PixRectMem object (see `PixRect.h` and `PixRectMem.h`). The PixRect is the same depth as the framebuffer.

---

**Note** – Currently, 2D `li3CopyFromDpBuffer()` is not called by the software pipeline.

---

## *li3CopyToDpBuffer() - 2D*

### **Syntax**

```
void XglDpCtx2d::li3CopyToDpBuffer(
    const Xgl_bounds_i2d*   src_rect,
    const Xgl_pt_i2d*       dest_pos,
    Xgl_buffer_sel          sel,
    const XglPixRectMem*    buf,
    Xgli_copy_to_dp_info*   copy_info);
```

### **Input Parameters**

<i>src_rect</i>	A rectangle in <i>buf</i> 's coordinates.
<i>dest_pos</i>	The position to copy to (relative to origin of the window).
<i>sel</i>	An image buffer in the device pipeline, when the device pipeline is multi-buffering, to copy into.
<i>buf</i>	The PixRect buffer where the data is copied from.
<i>copy_info</i>	Contains information about the incoming data such as the color map and color type of the data.

### **Attributes That the Device Pipeline Needs to Handle**

The Context attributes used by this function are:

```
XglContext::getRealPlaneMask()
XglContext::getRop()
```

### **What You Need to Know to Implement li3CopyToDpBuffer**

This function copies pixel data to the device pipeline's framebuffer out of memory. The memory is represent by a PixRectMem object (see `PixRect.h` and `PixRectMem.h`). The PixRect is the same depth as the frame buffer.

This function may be used to implement `xgl_copy_buffer()` in the future; review the comments in the definition of `Xgli_copy_to_dp_info` in the file `Li3Structs.h`.

## *li3CopyToDpBuffer() - 3D*

### **Syntax**

```
void XglDpCtx3d::li3CopyToDpBuffer(
    const Xgl_bounds_i2d*   src_rect,
    const Xgl_pt_i2d*       dest_pos,
    Xgl_buffer_sel          sel,
    const XglPixRectMem*    buf,
    Xgli_copy_to_dp_info*   copy_info);
```

### **Input Parameters**

<i>src_rect</i>	A rectangle in <i>buf</i> 's coordinates.
<i>dest_pos</i>	The position to copy to (relative to origin of the window).
<i>sel</i>	An image buffer in the device pipeline, when the device pipeline is multi-buffering, to copy into.
<i>buf</i>	The PixRect buffer where the data is copied from.
<i>copy_info</i>	Contains information about the incoming data such as the color map and color type of the data. See <code>Li2Structs.h</code> .

### **Attributes That the Device Pipeline Needs to Handle**

The Context attributes used by this function are:

```
XglContext::getBackgroundColor()
XglContext::getRealPlaneMask()
XglContext::getRop()
XglContext::getSurfFrontColor()
```

### **What You Need to Know to Implement 3D *li3CopyToDpBuffer***

This function copies pixel data to the device pipeline's frame buffer out of memory. The memory is represent by a PixRectMem object (see `PixRect.h` and `PixRectMem.h`). The PixRect is the same depth as the frame buffer. If the *copy\_info* pointer is NULL, the implementation of `li3CopyToDpBuffer()` operates as if a structure was given with `copy_info->do_zbuffer` set to FALSE and `copy_info->do_fill_style` set to FALSE.

The *Xgli\_copy\_to\_dp\_info* structure is used to provide information for `li3CopyToDpBuffer()`. The structure contains color map information for the source `PixRect` or raster; the pipeline needs to process this information. The structure also contains a flag to control whether the copy uses the Z-buffer. This flag will be `FALSE` for 2D Contexts but may be `TRUE` for 3D Contexts. In addition, the structure includes a flag for implementing fill style, but currently this flag will always be `FALSE`. See `Li3Structs.h` for comments in the definition of *Xgli\_copy\_to\_dp\_info*.

This function may be used to implement `xgl_copy_buffer()` in the future. Currently, for 3D, `li3CopyToDpBuffer()` is used by the accumulation operation and by `xgl_image()`.

## *li3MultiDot() - 2D*

### **Syntax**

```
void XglDpCtx2d::li3MultiDot(  
    const XglPrimData*  pd,  
    const Xgl_color*    color);
```

### **Input Parameters**

*pd*                      An XglPrimData object containing a list of point locations for the marker positions.

*color*                  The color value for the marker, if applicable.

### **Attributes That the Device Pipeline Needs to Handle**

The Context attributes used by this function are:

```
XglContext::getRealPlaneMask()  
XglContext::getRop()
```

### **What You Need to Know to Implement li3MultiDot**

This function draws a list of dots (i.e pixels) at the X,Y locations given in *pd*. If *color* is not NULL, then all of the dots are draw in that color. If it is NULL, then each dot is drawn in the color given by the per vertex color in *pd*.

## *li3MultiDot() - 3D*

### **Syntax**

```
void XglDpCtx3d::li3MultiDot(
    const XglPrimData*    pd,
    const Xgl_color*      color,
    Xgl_boolean*          picked);
```

### **Input Parameters**

*pd*                      Locations of the rendered dots.

*color*                  Color of the dots.

### **Output Parameter**

*picked*                TRUE if the primitive has been picked by Z-buffer-based picking.

### **Related Data Structures**

```
const Xgli_dot_control_3d& li3GetDotControl() const;

void li3SetDotControl(const Xgli_dot_control_3d&);

typedef struct {
    Xgl_boolean      do_aa;

    // This is ignored if do_aa is FALSE.
    Xgli_aa_info      aa_info;

    Xgl_usgn32        unused[4];
} Xgli_dot_control_3d;
```

### **Attributes That the Device Pipeline Needs to Handle**

The Context attributes used by this function are:

```
XglContext::getPickEnable()
XglContext::addPickToBuffer
XglContext::getPickId1()
XglContext::getPickId2()
XglContext::getBackgroundColor()
XglContext::getRealPlaneMask()
```



---

```
XglContext::getRenderBuffer()  
XglContext::getRop()  
XglContext3d::getBlendFreezeZBuffer()  
XglContext3d::getHlhrData()  
XglContext3d::getHlhrMode()  
XglContext3d::getZBufferCompMethod()  
XglContext3d::getZBufferWriteMask()
```

### ***What You Need to Know to Implement 3D li3MultiDot***

This function draws a list of dots at the X,Y locations given in *pd*. If *color* is not NULL, then all of the dots are draw in that color. If *color* is NULL, the each dot is drawn in the color given by the per vertex color in *pd*.

The control structure specifies if dots are antialiased. If they are, then a dot will touch more than one pixel.

## *li3Vector() - 2D*

### **Syntax**

```
Xgl_usgn32 XglDpCtx2d::li3Vector(
    const Xgli_vector_2d*    vector,
    const Xgl_color*         color);
```

### **Input Parameters**

**vector**                      Pointer to a structure defining the vector. Refer to the structure *Xgli\_vector\_2d* below.

**color**                        Color of the vector.

### **Related Data Structures**

```
const Xgli_vector_control_2d& li3GetVectorControl() const;

void li3SetVectorControl(
    const Xgli_vector_control_2d&);

typedef struct {
    Xgl_line_style          line_style; // style for vector
    const XglLinePattern*   pattern; // pattern to use
                                // for PATTERNED
                                // or ALT_PATTERNED
    const Xgl_color*        alt_color; // ALT_PATTERNED color
} Xgli_line_style_info;

typedef struct {
    Xgli_line_style_info     line_style_info;
    Xgl_usgn32               unused[4];
} Xgli_vector_control_2d;

typedef struct {
    Xgl_pt_i2d*              p1;          // end point 1
    Xgl_pt_i2d*              p2;          // end point 2
    Xgl_boolean              draw_last_pixel; // controls whether last
                                                // pixel is drawn.
    // the following is used for PATTERNED or ALT_PATTERNED vectors;
    Xgl_usgn32               pat_offset;   // pattern offset
} Xgli_vector_2d;
```

### ***Attributes That the Device Pipeline Needs to Handle***

The Context attributes used by this function are:

```
XglContext::getRealPlaneMask()  
XglContext::getRop()
```

### ***What You Need to Know to Implement li3Begin and li3End***

The `li3Vector()` function draws a vector from `vector->p1` to `vector->p2`. The function returns the number of pixels that will be drawn for the vector if it is not window clipped. This information is used by the software pipeline LI-2 to manage the pattern information for a polyline. If the flag `vector->draw_last_pixel` is `TRUE`, the whole vector is drawn, if it is `FALSE`, then the last pixel in the vector is not drawn.

The parameter *color* gives the color for the line and for the foreground pixels in an alt-patterned vector. The *line\_style\_info* argument controls whether the vector is solid, patterned or alt patterned. The *line\_style\_info->pattern* argument gives the pattern information.

## *li3Vector()* - 3D

### **Syntax**

```
Xgl_usgn32 XglDpCtx3d::li3Vector(
    const Xgli_vector_3d*    vector,
    const Xgl_color*        color,
    Xgl_boolean*            picked);
```

### **Input Parameters**

*vector*                      Pointer to a structure defining the vector. Refer to the structure *Xgli\_vector\_3d* below.

*color*                      Color of the vector.

### **Output Parameter**

*picked*                      TRUE if the primitive has been picked by Z-buffer-based picking.

### **Related Data Structures**

```
const Xgli_vector_control_3d& li3GetVectorControl() const;

void li3SetVectorControl(const Xgli_vector_control_3d&);

typedef struct {
    Xgli_line_style_info    line_style_info;
    Xgli_blend_type        blend_type;
    union {
        Xgli_transp_info    transp_info; // if a vector is
                                   // used to draw hollow;
                                   // it could be transparent.
        Xgli_aa_info        aa_info;
    } blend_info;
    Xgl_usgn32              unused[4];
} Xgli_vector_control_3d;
```

```

typedef struct {
    Xgl_pt_f3d*      p1;           // end point 1
    Xgl_pt_f3d*      p2;           // end point2
    Xgl_color*       p1_color;
    Xgl_color*       p2_color;
    Xgl_color*       p1_alt_color; // alt color for
                                // alt patterning
    Xgl_color*       p2_alt_color;
    Xgl_boolean      draw_last_pixel; // controls if last pixel
                                // is drawn.
    // the following is used for patterned vectors
    Xgl_usgn32        pat_offset;   // pattern offset
    Xgl_usgn32        unused[8];
} Xgli_vector_3d;

```

### ***Attributes That the Device Pipeline Needs to Handle***

The Context attributes used by this function are:

```

XglContext::getPickEnable()
XglContext::addPickToBuffer
XglContext::getPickId1()
XglContext::getPickId2()
XglContext::getBackgroundColor()
XglContext::getRealPlaneMask()
XglContext::getRenderBuffer()
XglContext::getRop()
XglContext3d::getBlendFreezeZBuffer()
XglContext3d::getHlhrsData()
XglContext3d::getHlhrsMode()
XglContext3d::getZBufferCompMethod()
XglContext3d::getZBufferWriteMask()

```

### ***What You Need to Know to Implement 3D Li3Vector***

This function draws a vector from *vector->p1* to *vector->p2*. The function returns the number of pixels would be drawn for the vector if it is not window clipped. This information is used by the software pipeline LI-2 to manage the pattern information for a polyline. If the flag *vector->draw\_last\_pixel* is TRUE, the whole vector is drawn; if it is FALSE, then the last pixel in the vector is not drawn.

The parameter `color` gives the color for the line and for the foreground pixels in an alternate patterned vector. If `color` is `NULL`, then `vector->p1_color` and `vector->p2_color` values are interpolated.

The `line_style_info` controls if the vector is solid, patterned or alt patterned. `line_style_info->pattern` gives the pattern information.

If the line style is alt-patterned and `vector->pt1_alt_color` and `vector->pt2_alt_color` are not `NULL`, then these colors are interpolated, and the interpolated color is used as the alternate pattern color. It is possible to interpolate the primary colors for the vector and use a constant alt color. In this case, `vector->pt1_alt_color` and `vector->pt2_alt_color` will be `NULL`, and the `line_style_info.alt_color` will be used.

The control structure allows for using vectors to implement transparent, hollow polygon edges, but we do not support this in XGL 3.0.1.

Vectors may be antialiased. The rule for determining if a vector is antialiased is:

```
// For now blending is only done when apiColorType is RGB
control.do_blend = ((vecCtrl.blend_info.aa_info.blend_eq !=
                    XGL_BLEND_NONE)
    && (vecCtrl.blend_info.aa_info.filter_width > 1)
    && (vecCtrl.blend_type == XGLI_BLEND_TYPE_AA)
    && (apiColorType == XGL_COLOR_RGB));
```

## *li3MultiSpan() - 2D*

### **Syntax**

```
void XglDpCtx2d::li3MultiSpan(
    const Xgli_span_list_2d* span_list,
    const Xgl_color* color);
```

### **Input Parameters**

*span\_list*                      Pointer to a structure defining the list of spans to be rendered. Refer to the structure *Xgli\_span\_list\_2d* below.

*color*                          Controls the color of the spans in the list.

### **Related Data Structures**

```
const Xgli_span_control_2d&li3GetSpanControl() const;
void li3SetSpanControl(const Xgli_span_control_2d&);

typedef struct {
    Xgl_surf_fill_style      fill_style;
    const XglRasterMem*      fill_raster;
    Xgl_pt_i2d               offset; // DC coord offset for
                                // realizing Fpat
                                // position attribute.
} Xgli_fill_style_info;

typedef struct {
    Xgli_fill_style_info      fill_style_info;
    Xgl_usgn32               unused[4];
} Xgli_span_control_2d;

typedef struct {
    Xgl_usgn32 num_x;
    Xgl_usgn32 y_start;
    Xgl_usgn32 x_start;
    Xgl_sgn32  x_delta; // either +1 or -1
    Xgl_color* color;
} Xgli_span_2d;

typedef struct {
    Xgl_usgn32 num_spans;
    Xgli_span_2d *spans;
} Xgli_span_list_2d;
```

### ***Attributes That the Device Pipeline Needs to Handle***

The Context attributes used by this function are:

```
XglContext::getRealPlaneMask()  
XglContext::getRop()  
XglContext::getBackgroundColor() (for opaque stipple filled  
patterns)
```

### ***What You Need to Know to Implement li3MultiDot***

This function draws a list of spans. A span is a horizontal run of pixels given by a starting X and Y location and the number of pixels to draw in the X direction. The X direction may be either to the left or to the right of the starting location.

The *color* parameter controls if all of the spans are drawn in the same color (then this parameter is not `NULL`) or if the color field in the span structure specifies the color for each span (then this parameter is `NULL`).

Spans can be filled with a pattern. The *fill\_style\_info* control fields specify the fill style for the spans and give the raster pattern to use for patterned spans.



## *li3MultiSpan() - 3D*

### **Syntax**

```
void XglDpCtx3d::li3MultiSpan(
    const Xgli_span_list_3d* span_list,
    const Xgl_color* color,
    Xgl_boolean* picked);
```

### **Input Parameters**

*span\_list*            Pointer to a structure defining the list of spans to be rendered. Refer to the structure *Xgli\_span\_list\_3d* below.

*color*                Controls the color of the spans in the list.

### **Output Parameter**

*picked*               TRUE if the primitive has been picked by Z-buffer-based picking.

### **Related Data Structures**

```
const Xgli_span_control_3d&li3GetSpanControl() const;

void li3SetSpanControl(const Xgli_span_control_3d&);

typedef struct {
    Xgli_fill_style_info            fill_style_info;

    Xgli_blend_type                blend_type;     // only NONE,
                                                    // SCREEN_DOOR,
                                                    // or TRANSP

    Xgli_transp_info                transp_info;

    Xgl_boolean                    do_texturing;
    Xgl_boolean                    do_lighting;
    Xgl_usgn32                     unused[4];
} Xgli_span_control_3d;

typedef struct {
    Xgl_usgn32                     num_x;

    Xgl_usgn32                     y_start;        // Y start value
    Xgl_usgn32                     x_start;        // X start value
```

```

Xgl_sgn32          x_delta;          // either +1 or -1
Xgli_fixed_z       z_start;          // Z start
Xgli_fixed_z       z_delta;          // Z increment
double             w_start;
double             w_delta;

/* These colors use Xgli_fixed_xy representation for indexed
   colors. The colors are interpolated in fixed point and LI3
   then truncates to an integer.
*/
Xgl_color          color_start;
Xgl_color          color_delta;

XgliUvSpanInfo3d   uv_info;
Xgl_usgn32         unused[8];
} Xgli_span_3d;

typedef struct {
    Xgl_usgn32       num_spans;
    Xgli_span_3d    *spans;

    Xgl_usgn32       unused[4];
} Xgli_span_list_3d;

```

## ***Attributes That the Device Pipeline Needs to Handle***

The Context attributes used by this function are:

```

XglContext::getPickEnable()
XglContext::addPickToBuffer
XglContext::getPickId1()
XglContext::getPickId2()
XglContext::getBackgroundColor()
XglContext::getRealPlaneMask()
XglContext::getRenderBuffer()
XglContext::getRop()
XglContext3d::getBlendFreezeZBuffer()
XglContext3d::getHlhrsData()
XglContext3d::getHlhrsMode()
XglContext3d::getZBufferCompMethod()
XglContext3d::getZBufferWriteMask()
XglContext3d::getDepthCueMode() (for texture mapping)

```

### ***What You Need to Know to Implement 3D `li3MultiSpan`***

This function draws a list of spans. A span is a horizontal run of pixels given by a starting X and Y location and the number of pixels to draw in the X direction. The X direction may be either to the left or to the right of the starting location.

The color parameters controls if all of the spans are drawn in the same color (then this parameter is not `NULL`) or if the color field in the span structure specifies the color for each span (then this parameter is `NULL`). If the color is given per span, then the color is interpolated using the *color\_start* and *color\_delta* fields.

When the color type is indexed and interpolation is being done, the colors in *Xgli\_span\_3d* are treated as fixed point numbers (*Xgli\_fixed\_xy* in `FixedPoint.h`). As an example, in *Xgli\_span\_3d*, *color\_start.index* should be cast to a *Xgli\_fixed\_xy*.

Spans can also be filled with a pattern. The *fill\_style\_info* control fields specifies the fill style for the spans and gives the raster pattern to use for patterned spans.

Spans can be rendered with transparency value and a transparency mode (either screen door or blended transparency); the field *blend\_type* in the the control structure manages blending.

In addition, spans can be filled with a texture-mapped pattern. If the *do\_texturing* field in *Xgli\_span\_control\_3d* is `TRUE`, spans are rendered with a texture-mapped pattern. The information needed to texture a span is passed from LI-2 in the *uv\_info* field of the *Xgli\_Span\_3d* structure. Texture mapping is implemented in `RefDpCtx`. If you choose not to use `RefDpCtx` but want to implement texture mapping, you can call the utility `XgliUtCalcTexturedColor`. See Chapter 10, “Utilities”.

XGL uses hyperbolic interpolation to arrive at an intermediate (u,v) in a span. The class `XgliUvSpanInfo3d` encapsulates the Data Map Texture object (u,v) numerator, denominator, (u,v) deltas, the start MipMap level, and the delta for the span. In addition, it has the lighting coefficients that are used if lighting is applicable.

`XgliUtUvSpanInfo3d` provides functions to retrieve this information and increment the information as the span is traversed. The interfaces provided by this class are:

```
void setNumInfo(Xgl_usgn32 n)
```

This function sets the number of texture coordinates (u,v) and related information that needs to be stored in the class. (This corresponds to the number of data maps that are active). This function allocates the necessary space for the storage.

```
Xgl_usgn32getNumInfo() const
```

This function returns the number of sets of texture coordinate ({u,v}) values.

```
Xgli_light_and_uv_info* getLightAndUvInfo()
```

This function returns a structure that contains the {u,v} related fields such as numerator and delta (for hyperbolic interpolation), the start mipmap level and delta for the span as well as lighting coefficients and delta for the span. The function is called when various fields need to be filled in.

```
void getPixelDataInfo(Xgli_pixel_data_info*) const
```

This function takes the current value of texture coordinates ({u,v}) and light coefficients at a pixel location, does a perspective divide, and returns the values. The value returned is the texture coordinate ({u,v}) that is used to look up in the texture map, and the lighting coefficients used to light the pixel. Note that since there can be multiple data maps (and several textures within a data map) that are active. The structure has an array of {u,v} corresponding to the number of data map objects that are active.

```
void incrementLightAndUvInfo()
```

This function increments the pixel information as it proceeds along the span. Typically, for each pixel the caller uses `getPixelDataInfo()` to get the {u,v} and lighting values for that pixel and then increments the pixel information to reflect the correct values for the next pixel in the span.

This chapter provides information on XGL utilities. XGL utilities are designed to perform specific operations and are useful for special case processing. Utility classes have `Ut` in the name, for example `XgliUtFooBar`. The utilities are part of the core XGL library; they are not in a separately loaded library.



Most XGL utilities are found in these header files:

- `RefDpCtx.h`, `RefDpCtx2d.h`, and `RefDpCtx3d.h`
- `CheckBbox.h`
- `CopyBuffer.h`
- `PgonClass.h`
- `Utils3d.h`
- `utils.h`

## *RefDpCtx*

RefDpCtx (Reference Device Pipeline Context) is a utility object that provides a non-optimized implementation of LI-3 functions and several LI-1 pixel functions for the device pipeline. Each device pipeline must implement the LI-3 functions on its device. However, the pipeline can choose to use the RefDpCtx LI-3 implementation of the LI-3 functions.

The way a device is described to the RefDpCtx object is through a number of PixRect objects. These PixRect objects are abstractions of the buffers managed by the device, for example, the image buffer, Z-buffer, and accumulation buffer. The RefDpCtx object performs all operations for rendering at the LI-3 level, including texture mapping, blending, and transparency. RefDpCtx uses the methods of the PixRect object to read and write pixels.

Before you use RefDpCtx, you should consider the following design issues:

- Because RefDpCtx accesses the hardware via PixRect objects, the pipeline must bracket calls to RefDpCtx with `WIN_LOCK()` and `WIN_UNLOCK()` calls. It is up to the pipeline to manage window locking around a RefDpCtx call.
- The pipeline must communicate certain attribute changes to RefDpCtx. The device pipeline itself receives information about attribute changes through the device pipeline context object (`objectSet()`) at the various LI layers, and it handles those changes internally. However, the RefDpCtx is a separate utility object for the pipeline; therefore, the pipeline needs to pass along information about attribute changes to this object as follows:
  - The pipeline can determine itself whether specific attributes have changed or whether the Device's color map object changed. If either of these changed, the pipeline can call RefDpCtx interfaces `generalGroupChanged()` or `cmapChanged()` to tell RefDpCtx that changes have occurred.
- RefDpCtx uses PixRects to represent the buffers of the device. PixRects provide subclasses to handle memory-mapped buffers. If your device's buffers are not memory-mappable, you will have to create your own subclass of PixRect to communicate with your device. See Chapter 4, "Internal Data Storage" for information on PixRect objects.

## Using RefDpCtx

To use RefDpCtx for 2D, the pipeline needs a PixRect object to represent the image buffer (or the current image buffer if multibuffering is in effect). For 3D, the pipeline needs PixRect objects to represent the image buffer, the Z-buffer, and the accumulation buffer.

To make the PixRects available to the RefDpCtx object, users of RefDpCtx2d will call:

`setImagePixRect()` Assigns a PixRect for the image buffer.

`setClipMaskPixRect()` Assigns a Pixrect for the clip mask.

Users of RefDpCtx3d will call the above functions and the following:

`setZbufferPixRect()` Assigns a PixRect for the Z-buffer.

`setAccumBufferPixRect()` Assigns a PixRect for the accumulation buffer.

The example code below shows how a 3D pipeline might create RefDpCtx object in its XglDpCtx class.

```
XglDpCtx3dCfb::XglDpCtx3dCfb(XglDpDevCfb* dD,
                             XglContext3d* ctx) :
    XglDpCtx3d(ctx),
    refDpCtx((XglRaster*)dD->getDevice(), ctx)
{
    dpDev = dD;
    drawable = dpDev->getDevice()->getDrawable();

    // the following XglDpDev functions are device-dependent
    functions
    // that return pointers to PixRects
    refDpCtx.setImagePixRect(dpDev->getWinPixRect());
    refDpCtx.setZbufferPixRect(dpDev->getZbufferPixRect());
    refDpCtx.setAccumBufferPixRect(dpDev-
    >getAccumBufferPixRect());
}
```

Once the PixRects are assigned to the RefDpCtx, the pipeline can use them to render LI-3 functions. There is data associated with the RefDpCtx object, such as plane mask, ROP, and Z-buffer compare method; therefore, the pipeline must check the relevant state and update the RefDpCtx object if necessary.

The example code below shows a 3D pipeline implementing LI-3 using RefDpCtx LI-3 calls. Note that the calls to RefDpCtx rendering functions must be bracketed by calls to lock and unlock the clip list. RefDpCtx calls the PixRect functions `getValue()` and `setValue()` to modify the pixel values. You can copy or modify this source code sample as long as the resulting code is used to create a loadable pipeline for XGL.

```
const Xgli_span_control_3d&
XglDpCtx3dCfb::li3GetSpanControl() const
{
    return refDpCtx.li3GetSpanControl();
}

void XglDpCtx3dCfb::li3SetSpanControl(const
Xgli_span_control_3d&
                                   sc)
{
    refDpCtx.li3SetSpanControl(sc);
}

void XglDpCtx3dCfb::li3MultiSpan(
    const Xgli_span_list_3d* span_list,
    const Xgl_color*color,
    int* picked)
{
    // Update RefDpCtx with attribute changes

    WIN_LOCK(drawable);

    // Handle window obscured or moved

    refDpCtx.li3MultiSpan(span_list, color, picked);

    WIN_UNLOCK(drawable);
}
```



## RefDpCtx Interfaces



The RefDpCtx classes include LI-1 and LI-3 functions; documentation on those functions is provided in Chapter 9, “Writing Loadable Interfaces”. See the header files `RefDpCtx.h`, `RefDpCtx2d.h`, and `RefDpCtx3d.h` for a complete list of RefDpCtx interfaces.

The following functions are unique to RefDpCtx and its subclasses.

```
void setImagePixRect(XglPixRect*)
```

Sets the PixRect that represents the image buffer to draw into. If single buffering is being used, this PixRect will be set once; if multi-buffering is used, this PixRect will be changed.

```
void setClipMaskPixRect(XglPixRectMem* i,  
                        Xgl_boolean no_need_to_clip)
```

Call when the clip list changes. The PixRect for the clip area is a 1-bit deep pixrect that represents the mask for the clip area. This PixRect comes from the XglDrawable function `getMergeClipMask()`.

```
void syncClipMask()
```

Gets the current clip mask from the Drawable. In the current implementation, `syncclipMask()` is called internally to ensure that the current clip mask is always up to date.

```
void setDoMaskAndRop(Xgl_boolean)
```

Controls whether RefDpCtx does the plane mask and rop. If it returns `TRUE`, the current plane mask and rop are used in calculating the pixel value. If it returns `FALSE`, then the plane mask and rop are not applied.

```
void cmapChanged()
```

When `XGL_CTX_DEVICE` is passed through `objectSet()`, the device pipeline should call this function to inform RefDpCtx that the Device's Color Map object has changed.

```
void generalGroupChanged()
```

When `XGL_CTX_PLANE_MASK` (2D and 3D), `XGL_CTX_ROP` (2D and 3D), and `XGL_3D_CTX_Z_BUFFER_COMP_METHOD` (3D only) are passed through `objectSet()`, the device pipeline should call this function to inform `RefDpCtx` that changes have occurred in plane mask, ROP, or Z-buffer compare method.

```
void setZbufferPixRect(XglPixRect* z)
```

Sets the `PixRect` that represents the Z-buffer to hold Z values.

```
void setAccumBufferPixRect(XglPixRect* a)
```

Used by copy buffer during the accumulation operation.

## 3D Utilities

XGL utilities for 3D operations are in the header file `Utils3d.h`.

### *XgliUtAccumulate*

```
void XgliUtAccumulate(
    const XglPixRectMem* src_buf,
    const Xgl_bounds_i2d* rect,
    float src_wt,
    float dst_wt,
    XglPixRectMem* dst_buf,
    const Xgl_pt_i2d* dst_pos)
```

Accumulates from the source buffer *src\_buf* to the destination buffer *dst\_buf*. *rect* and *src\_wt* apply to the source buffer. *pos* and *dst\_wt* apply to the destination buffer.

#### **Input Parameters**

<i>src_buf</i>	The source buffer used in the accumulation operation. The source buffer should be a 32-bit pixrect.
<i>rect</i>	The rectangle in the source buffer that needs to be accumulated.
<i>src_wt</i>	The source weight in the accumulation operation.
<i>dst_wt</i>	The destination weight in the accumulation operation.

*dst\_pos*                      The position in the destination buffer to be used as starting position.

**Output Parameter**

*dst\_buf*                      The destination buffer in the accumulation operation. The destination buffer is either a 32-bit or 48-bit Pixrect.

## *XgliUtCdAnnCircleApprox*

```
Xgl_sgn32 XgliUtCdAnnCircleApprox(  
    XglContext3d      *ctx,  
    XglConicList3d    *circle_list)
```

Evaluates the number of points to be used to approximate an annotation circle when the value of the attribute XGL\_CTX\_NURBS\_CURVE\_APPROX is one of the following:

```
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
```

**Input Parameters**

*ctx*                          Pointer to a 3D Context.

*circle\_list*                Pointer to an XglConicList3d object containing a list of circles or circular arcs.

**Output Parameter**

None

**Return Value**

Returns the number of points to be used to approximate an annotation circle.

## *XgliUtAnnCircleApprox*

```
Xgl_sgn32 XgliUtAnnCircleApprox(
    XglContext3d      *ctx,
    Xgl_circle_list   *circle_list)
```

See *XgliUtCdAnnCircleApprox* for a description of the functionality.

### ***Input Parameters***

*ctx*                      Pointer to a 3D Context.  
*circle\_list*            Pointer to a list of circles.

### ***Output Parameter***

None

### ***Return Value***

Returns the number of points to be used to approximate an annotation circle.

## *XgliUtAnnArcApprox*

```
Xgl_sgn32 XgliUtAnnArcApprox(
    XglContext3d      *ctx,
    Xgl_arc_list      *arc_list)
```

See *XgliUtAnnArcApprox* for a description of the functionality.

### ***Input Parameters***

*ctx*                      Pointer to a 3D Context.  
*arc\_list*                Pointer to a list of arcs.

### ***Output Parameter***

None

### ***Return Value***

Returns the number of points to be used to approximate an annotation arc.

## *XgliUtCdAnnEllArcApprox*

```
Xgl_sgn32 XgliUtCdAnnEllArcApprox(  
    XglContext3d      *ctx,  
    XglConicList3d    *ell_list)
```

Evaluates the number of points to be used to approximate an annotation ellipse when the value of the attribute XGL\_CTX\_NURBS\_CURVE\_APPROX is one of the following:

```
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
```

### ***Input Parameters***

<i>ctx</i>	Pointer to a 3D context.
<i>ell_list</i>	Pointer to an XglConicList3d object containing a list of elliptical arcs.

### ***Output Parameter***

None

### ***Return Value***

Returns the number of points to be used to approximate an annotation ellipse.

## *XgliUtAnnEllArcApprox*

```
Xgl_sgn32 XgliUtAnnEllArcApprox(  
    XglContext3d      *ctx,  
    Xgl_ell_list      *ell_list)
```

See XgliUtAnnEllArcApprox for a description of the functionality.

### ***Input Parameters***

<i>ctx</i>	Pointer to a 3D context.
<i>ell_list</i>	Pointer to a list of ellipses.

### ***Output Parameter***

None

### ***Return Value***

Returns the number of points to be used to approximate an annotation ellipse.

## ***XgliUtCalcDcueIndex***

```
void XgliUtCalcDcueIndex(
    XglContext3d*      ctx3d,
    XglViewGrp3dItf*   view_itf,
    Xgl_color*         color_in,
    float              z,
    Xgl_color*         color_out)
```

Used when the color type is XGL\_COLOR\_INDEX and thus expects colors in the INDEX format. The function depth cues a input color *color\_in* given the Z (in DC) value at which to depth cue the color.

### ***Input Parameters***

<i>ctx3d</i>	The Context used in rendering the primitive.
<i>view_itf</i>	The view group interface from which the depth cue planes in DC and the DC viewport is used in calculating the depth cue color.
<i>color_in</i>	The color to be depth cued
<i>z</i>	The Z value (in DC) at which to depth cue.

### ***Output Parameter***

<i>color_out</i>	The depth cued color
------------------	----------------------

## *XgliUtCalcDcueRgb*

```
void XgliUtCalcDcueRgb(  
    XglContext3d*      ctx3d,  
    XglViewGrp3dItf*  view_itf,  
    Xgl_color*         color_in,  
    float              z,  
    Xgl_color*         color_out)
```

Used when the color type is `XGL_COLOR_RGB` and thus expects colors in the RGB format. The function depth cues a input color *color\_in* given the Z (in DC) value at which to depth cue the color.

### ***Input Parameters***

<i>ctx3d</i>	The Context used in rendering the primitive.
<i>view_itf</i>	The view group interface from which the depth cue planes in DC and the DC viewport is used in calculating the depth cue color.
<i>color_in</i>	The color to be depth cued.
<i>z</i>	The Z value in DC at which to depth cue.

### ***Output Parameter***

<i>color_out</i>	The depth cued color
------------------	----------------------

## *XgliUtCalcDoubleCircle*

```
void XgliUtCalcDoubleCircle(  
    float      *mem,  
    Xgl_sgn32  n_steps,  
    float      d_angle)
```

Calculates the points (x, y) on the unit circle which subdivides the unit circle into (n\_steps - 1) segments. The calculated points (x,y) are stored twice in the array mem. The first copy of the points (x,y) is stored in:

```
(mem[0], mem[2*n_steps]), (mem[1], mem[2*n_steps+1]), ... ,  
(mem[n_steps-2], mem[3*n_steps-2]), (mem[n_steps-1], mem[3*n_steps -1])
```

The second copy of the points is stored in:

```
(mem[n_steps], mem[3*n_steps]), (mem[n_steps+1], mem[3*n_steps+1]),... ,
(mem[2*n_steps-2], mem[4*n_steps-2]), (mem[2*n_steps-1], mem[4*n_steps-1])
```

### ***Input Parameters***

***n\_steps*** An integer indicating that the unit circle is subdivided into (n\_steps - 1) segments.

***d\_angle*** The angle in radian formed by two consecutive subdivision points on the unit circle with the center of the unit circle.

### ***Output Parameter***

***mem*** An array of floats allocated by the caller. The size of the array is 4\*n\_steps. This array will hold the points calculated by this utility.

## ***XgliUtCalcLightingCompRgb***

```
void XgliUtCalcLightingCompRgb(
    XglContext3d*      ctx,
    XglViewGrp3dItf*   view_itf,
    Xgl_pt_f3d*         normal,
    Xgl_pt_f3d*         point,
    const Xgli_surf_attr_3d* surf,
    Xgl_boolean         front_flag,
    Xgl_color*          comp_A,
    Xgl_color*          comp_B)
```

This routine takes an input point and normal and calculates the two color components necessary for texture mapping at that point. Consult the texture mapping documentation for more details regarding the color components and their use. This routine can only be used if the XGL color type is XGL\_COLOR\_RGB.

### ***Input Parameters***

***ctx*** Context containing light sources and lighting parameters.

***view\_itf*** View group interface used to obtain transformed light positions/directions.



<i>normal</i>	Input facet or vertex normal (depending on whether the illumination is per_vertex, or per_facet, respectively).
<i>point</i>	Input 3D point.
<i>surf</i>	Surface attributes, either front or back.
<i>front_flag</i>	Flag indicating whether the normal is front facing.

### **Output Parameters**

<i>comp_A</i> ; <i>comp_B</i>	Lighting components to be used during scan conversion by the texture mapping code. <i>comp_A</i> is the color scale factor, while <i>comp_B</i> is the offset.
-------------------------------	--

## ***XgliUtCalcLightingRgb and XgliUtCalcLightingIndex***

```
void XgliUtCalcLighting{Rgb,Index}(
    XglContext3d*      ctx,
    XglViewGrp3dItf*   view_itf,
    Xgl_color*          color_in,
    Xgl_pt_f3d*         normal,
    Xgl_pt_f3d*         point,
    const Xgli_surf_attr_3d* surf,
    Xgl_boolean         front_flag,
    Xgl_color*          color_out)
```

These routines apply the current light sources and lighting parameters to the input color, normal, and 3D point and returns a new, calculated color.

*XgliUtCalcLightingRgb* can only be used if the XGL color type is `XGL_COLOR_RGB`. The corresponding utility *XgliUtCalcLightingIndex* is used in the case of `XGL_COLOR_INDEX`.

### **Input Parameters**

<i>ctx</i>	XGL Context containing light sources and lighting parameters.
<i>view_itf</i>	View group interface used to obtain transformed light positions/directions.
<i>color_in</i>	Input color.
<i>normal</i>	Input facet or vertex normal (depending on whether the illumination is per_vertex, or per_facet, respectively).
<i>point</i>	Input 3D point.

*surf*                      Surface attributes, either front or back.  
*front\_flag*              Flag indicating whether the normal is front facing.

**Output Parameter**

*color\_out*              The output color, adjusted for the Context lighting values.

***XgliUtCalcSingleCircle***

```
void XgliUtCalcSingleCircle(
    float          *mem,
    Xgl_sgn32      n_steps)
```

Calculates the points (x,y) on the unit circle which subdivides the unit circle into (n\_steps - 1) equal segments. The calculated points (x,y) are stored in the array mem in the following way:

```
(mem[0], mem[n_steps]), (mem[1], mem[n_steps+1]), ...,
(mem[n_steps-2], mem[2*n_steps-2]), (mem[n_steps-1], mem[2*n_steps-1])
```

**Input Parameters**

*n\_steps*              An integer indicating that the unit circle is subdivided into (n\_steps - 1) segments.

**Output Parameter**

*mem*                    An array of floats allocated by the caller. The size of the array is 2\*n\_steps. This array will hold the points calculated by this utility.

## *XgliUtCalcTexturedColor*

```
void XgliUtCalcTexturedColor(  
    XglContext3d*      ctx,  
    const XgliUvSpanInfo3d* data_info,  
    Xgl_color*         obj_clr,  
    Xgl_boolean        do_lighting,  
    Xgl_usgn32         z,  
    Xgl_color*         color_out)
```

Applies the texture maps in the current *ctx*. does lighting and depth cueing and returns the textured pixel. The caller passes as input the texture coordinate (u, v) of the pixel and the lighting components at the pixel (these are encapsulated in *data\_info*). Thus, this utility does texture lookup and interpolation based on the (u, v) value, followed by color composition of the texel with the object color *obj\_clr* to obtain the textured color. It also does lighting and depth cueing if applicable.

### ***Input Parameters***

<i>ctx</i>	Context whose textures are applied.
<i>data_info</i>	Contains the texture coordinate (u, v) (before the divide by 1/w) as well as the lighting components at that pixel.
<i>obj_clr</i>	Pixel color before the texturing operation. This is the intrinsic color of the pixel.
<i>do_lighting</i>	If TRUE, lighting is performed.
<i>z</i>	Z value in DC of the pixel, used when performing depth cueing.

### ***Output Parameter***

<i>color_out</i>	The output color after textures have been applied and lighting and depth cueing have been performed.
------------------	--

## *XgliUtCalc3dTriOrientation*

```
int XgliUtCalc3dTriOrientation(
    Xgl_pt_f3d*    v1,
    Xgl_pt_f3d*    v2,
    Xgl_pt_f3d*    v3,
    Xgl_pt_f3d*    fn)
```

Provides the winding of the points of a triangle given its three vertices and the facet normal.

### ***Input Parameters***

<i>v1</i>	Coordinates of vertex 1.
<i>v2</i>	Coordinates of vertex 2.
<i>v3</i>	Coordinates of vertex 3.
<i>fn</i>	Facet normal of the face.

### ***Output Parameter***

None.

### ***Return Value***

Returns the orientation, which can be either XGLI\_PGON\_ORIENT\_CW or XGLI\_PGON\_ORIENT\_CCW.

## *XgliUtComputeColorComp*

```
void XgliUtComputeColorComp(
    Xgli_acolor*    tex_clr,
    Xgl_color*      obj_clr,
    Xgl_texture_desc* tex_desc,
    Xgl_color*      out_clr)
```

Takes an incoming color *obj\_clr* and combines it with the texel *tex\_clr* in a manner described in the *tex\_desc*. The result of this color composing is returned in *out\_clr*.

**Input Parameters**

<i>tex_clr</i>	The texel value that should be used in color composition.
<i>obj_clr</i>	The object color that should be combined with the texel.
<i>tex_desc</i>	The texture descriptor that contains the color composition method to use.

**Output Parameter**

<i>out_clr</i>	The output color after color composition.
----------------	---

***XgliUtComputeColorInterp***

```
void XgliUtComputeColorInterp(  
    Xgli_pt_uv_info*      pdata,  
    Xgl_texture_desc*     tex_desc,  
    Xgli_acolor*          texel)
```

Takes as input the texture coordinate (u, v) and the MipMap level (in which this pixel is located) encapsulated in *pdata* and the texture descriptor *tex\_desc* that should be used to do the texture lookup and interpolation to obtain the texture value.

**Input Parameters**

<i>pdata</i>	Contains the texture coordinate (u, v) and the MipMap level of the pixel.
<i>tex_desc</i>	The texture descriptor that should be used for lookup and interpolation.

**Output Parameter**

<i>texel</i>	The output color after applying lookup and interpolation. Note that the type is <i>Xgli_acolor</i> ; thus the returned value will have an alpha value as well.
--------------	--

## *XgliUtComputeDiffuseColor*

```
void XgliUtComputeDiffuseColor(
    Xgli_pixel_data_info* pdata,
    Xgl_color*           in_clr,
    Xgl_color*           out_clr)
```

Takes the intrinsic color *in\_clr* and applies the texture maps that apply to the diffuse component. This involves texture lookup and interpolation to obtain the texture value, composing the *in\_clr* with the texture color to obtain the *out\_clr*.

### **Input Parameters**

*pdata*                      This structure contains the texture maps that are active and the associated texture coordinates (u, v).

*in\_clr*                     Pixel color before the texturing operation. This is the intrinsic color of the pixel.

### **Output Parameter**

*out\_clr*                    The output color (diffuse color) after applying the textures that affect the diffuse component of the rendering pipeline.

## *XgliUtComputeFinalColor*

```
void XgliUtComputeFinalColor(
    Xgli_pixel_data_info* pdata,
    Xgl_color*           in_clr,
    Xgl_color*           out_clr)
```

Takes the depth cued color *in\_clr* and applies the texture maps that apply to the final (after depth cueing) component. This involves texture lookup and interpolation to obtain the texture value, composing the *in\_clr* with the texture color to obtain the *out\_clr*.

### **Input Parameters**

*pdata*                      Contains the texture maps that are active and the associated texture coordinates (u, v).

*in\_clr*                     Pixel color before the texturing operation. This is the depth cued color of the pixel.

**Output Parameter**

*out\_clr*                      The output color after applying the textures that affect the final component of the rendering pipeline.

***XgliUtComputeFn***

```
int XgliUtComputeFn(
    Xgl_operator      op,
    Xgl_geom_normal   geom_normal,
    Xgl_boolean        normalize,
    Xgl_usgn32         row_dim,
    Xgl_usgn32         col_dim,
    Xgl_usgn32         num_pt_lists,
    Xgl_pt_list*       pl,
    Xgl_pt_f3d*        facet_normal)
```

Computes the facet normal and returns the normals. For surfaces other than quadmesh, the *row\_dim* and *col\_dim* are ignored. The utility provides the option for normalizing.

**Input Parameters**

<i>op</i>	Type of operator
<i>geom_normal</i>	Geometry normal format as defined by the API attribute XGL_3D_CTX_SURF_GEOM_NORMAL.
<i>normalize</i>	If TRUE, normalizes the normal.
<i>row_dim</i>	Number of rows when <i>op</i> is XGL_OP_XGL_QUADRILATERAL_MESH. This parameter is ignored for other <i>ops</i> .
<i>col_dim</i>	Number of columns when <i>op</i> is XGL_OP_XGL_QUADRILATERAL_MESH. This parameter is ignored for other <i>ops</i> .
<i>num_pt_lists</i>	Number of point lists in <i>pl</i> . For triangle strip and quadmesh, <i>num_pt_lists</i> is assumed to be 1, so its value is ignored.
<i>pl</i>	Geometry information describing the primitive.

**Output Parameter**

*facet\_normal* Caller allocated structure in which to return the computed normals. The memory to be allocated by the caller depends on the primitive:

Multisimple polygon = num\_pt\_lists \*  
sizeof(Xgl\_pt\_f3d)

Triangle strip = (pl[0].num\_pts - 2)  
\*sizeof(Xgl\_pt\_f3d)

Quadmesh = (row\_dim - 1)\*(col\_dim-1) \*  
sizeof(Xgl\_pt\_f3d)

Polygon = sizeof(Xgl\_pt\_f3d)

**Return Value**

Returns a 1 if the normal was computed successfully; otherwise, returns a 0.

***XgliUtComputeFnReverse***

```
int XgliUtComputeFnReverse(
    Xgl_facet_list* fl,
    Xgl_usgn32 num_facets,
    Xgl_pt_f3d* fn)
```

Reverses the facet normals using the normals in *fl*.

**Input Parameters**

*fl* Input from which the facet normals should be reversed.

*num\_facets* Number of facets or number of facet normals.

**Output Parameter**

*fn* Caller allocated structure in which to return the facet normals. The memory to be allocated is:  
num\_facets \* sizeof (Xgl\_pt\_f3d)

**Return Value**

Returns a 1 if the function is successful; otherwise, returns a 0.



## ***XgliUtComputeIndepTriFn***

```
int XgliUtComputeIndepTriFn(  
    Xgl_geom_normal    geom_normal,  
    Xgl_boolean        normalize,  
    Xgl_pt_list*       pl,  
    Xgl_pt_f3d*        facet_normal)
```

Computes the normal for the facets in a set of independent triangles (a subset of the triangle list primitive) from the point list and returns the computed normals.

### ***Input Parameters***

<i>geom_normal</i>	Geometry normal format as defined by the API attribute XGL_3D_CTX_SURF_GEOM_NORMAL
<i>normalize</i>	If TRUE, normalizes the normal.
<i>pl</i>	Data from which the normal should be calculated.

### ***Output Parameter***

<i>facet_normal</i>	Caller allocated structure in which to return the computed normals. The memory to be allocated is: ( <i>pl</i> [0].num_pts - 2) * sizeof (Xgl_pt_f3d)
---------------------	--

### ***Return Value***

Returns a 1 if the normal was computed successfully; otherwise, returns a 0.

## ***XgliUtComputeIndepTriFnPl***

```
int XgliUtComputeIndepTriFnPl(  
    Xgl_geom_normal    geom_normal,  
    Xgl_boolean        normalize,  
    Xgl_pt_list*       pl,  
    Xgl_tlist_flags    tlist_flags,  
    Xgl_pt_f3d*        facet_normal)
```

Computes the normal for the facets in a set of independent triangles (a subset of the triangle list primitive) from an input point list provided by the user and returns the computed normals.

### **Input Parameters**

<i>geom_normal</i>	Geometry normal format as defined by the API attribute XGL_3D_CTX_SURF_GEOM_NORMAL.
<i>normalize</i>	If TRUE, normalizes the normal.
<i>pl</i>	Vertex data.
<i>tlist_flags</i>	Global triangle list flags which were passed into the <code>xgl_triangle_list()</code> primitive.

### **Output Parameter**

<i>facet_normal</i>	Caller allocated structure in which to return the computed normals. The memory to be allocated is: ( <code>pl[0].num_pts - 2</code> ) * <code>sizeof (Xgl_pt_f3d)</code>
---------------------	---

### **Return Value**

Returns a 1 if the normal was computed successfully; otherwise, returns a 0.

## ***XgliUtComputeMspFn***

```
int XgliUtComputeMspFn(
    Xgl_geom_normal    geom_normal,
    Xgl_boolean        normalize,
    Xgl_usgn32         num_pt_lists,
    Xgl_pt_list*       point_list,
    Xgl_pt_f3d*        facet_normal)
```

Computes the normals of the simple polygon in a multisimple polygon call from the *point\_list* and returns the computed normal.

### **Input Parameters**

<i>geom_normal</i>	Geometry normal format as defined by the API attribute XGL_3D_CTX_SURF_GEOM_NORMAL.
<i>normalize</i>	If TRUE, normalizes the normal.
<i>num_pt_lists</i>	Number of point lists in <i>point_list</i> .
<i>point_list</i>	Data from which the normal should be calculated.

**Output Parameter**

*facet\_normal* Caller allocated structure in which to return the computed normals. The memory to be allocated is:  
`num_pt_lists * sizeof (Xgl_pt_f3d)`

**Return Value**

Returns a 1 if the normal was computed successfully; otherwise, returns a 0.

***XgliUtComputePolygonFn***

```
int XgliUtComputePolygonFn(
    Xgl_geom_normal    geom_normal,
    Xgl_boolean        normalize,
    Xgl_usgn32         num_pt_lists,
    Xgl_pt_list*       point_list,
    Xgl_pt_f3d*        facet_normal)
```

Computes the normal for the polygon from the point list. The first non-degenerate boundary of the polygon is used in the normal computation.

**Input Parameters**

*geom\_normal* Geometry normal format as defined by the API attribute  
`XGL_3D_CTX_SURF_GEOM_NORMAL`

*normalize* If TRUE, normalizes the normal.

*num\_pt\_lists* Number of point lists in *point\_list*.

*point\_list* Data from which the normal is calculated.

**Output Parameter**

*facet\_normal* Caller allocated structure in which to return the computed normals. The memory to be allocated is  
`sizeof (Xgl_pt_f3d).`

**Return Value**

Returns a 1 if the normal was computed successfully; otherwise, returns a 0.

## ***XgliUtComputeQuadMeshFn***

```
int XgliUtComputeQuadMeshFn(
    Xgl_geom_normal    geom_normal,
    Xgl_boolean        normalize,
    Xgl_usgn32         row_dim,
    Xgl_usgn32         col_dim,
    Xgl_pt_list*       point_list,
    Xgl_pt_f3d*        facet_normal)
```

Computes the normal for the facets in the quadrilateral mesh from the point data in *point\_list* and returns the computed normals.

### ***Input Parameters***

<i>geom_normal</i>	Geometry normal format as defined by the API attribute XGL_3D_CTX_SURF_GEOM_NORMAL.
<i>normalize</i>	If TRUE, normalizes the normal
<i>point_list</i>	Data from which the normal is calculated.

### ***Output Parameter***

<i>facet_normal</i>	Caller allocated structure in which to return the computed normals. The memory to be allocated is: (row_dim - 1) * (col_dim - 1) * sizeof(Xgl_pt_f3d)
---------------------	---

### ***Return Value***

Returns a 1 if the normal was computed successfully; otherwise, returns a 0.

## ***XgliUtComputeReflectedColor***

```
void XgliUtComputeReflectedColor(
    Xgli_pixel_data_info* pdata,
    Xgl_color*           in_clr,
    Xgl_color*           out_clr)
```

Takes the diffuse color *in\_clr* and applies the texture maps that apply to the reflected component (after lighting). Applying the texture maps involves texture lookup and interpolation to obtain the texture value, composing the *in\_clr* with the texture color to obtain the *out\_clr*.

**Input Parameters**

<i>pdata</i>	Contains the texture maps that are active and the associated texture coordinates (u, v).
<i>in_clr</i>	Pixel color before the texturing operation. This is the lit color of the pixel.

**Output Parameter**

<i>out_clr</i>	The output color after applying the textures that affect the reflected component of the rendering pipeline.
----------------	---

***XgliUtComputeTstripFn***

```
int XgliUtComputeTstripFn(  
    Xgl_geom_normal    geom_normal,  
    Xgl_boolean        normalize,  
    Xgl_pt_list*       point_list,  
    Xgl_pt_f3d*        facet_normal)
```

Computes the normal for the facets in the triangle strip.

**Input Parameters**

<i>geom_normal</i>	Geometry normal format as defined by the API attribute XGL_3D_CTX_SURF_GEOM_NORMAL
<i>normalize</i>	If TRUE, normalizes the normal.
<i>point_list</i>	Data from which the normal is calculated.

**Output Parameter**

<i>facet_normal</i>	Caller allocated structure in which to return the computed normals. The memory to be allocated is: (pl[0].num_pts - 2) * sizeof (Xgl_pt_f3d)
---------------------	---

**Return Value**

Returns a 1 if the normal was computed successfully; otherwise, returns a 0.

## *XgliUtComputeTstripFnPl*

```
int XgliUtComputeTstripFnPl(
    Xgl_geom_normal    geom_normal,
    Xgl_boolean        normalize,
    Xgl_pt_list*       point_list,
    Xgl_tlist_flags    tlist_flags,
    Xgl_pt_f3d*        facet_normal)
```

Computes the normal for the facets in a triangle strip from an input point list and returns the computed normals.

### ***Input Parameters***

<i>geom_normal</i>	Geometry normal format as defined by the API attribute XGL_3D_CTX_SURF_GEOM_NORMAL
<i>normalize</i>	If TRUE, normalizes the normal.
<i>point_list</i>	Vertex data.
<i>tlist_flags</i>	Global triangle list flags that were passed into the <code>xgl_triangle_list()</code> primitive.

### ***Output Parameter***

<i>facet_normal</i>	Caller allocated structure in which to return the computed normals. The memory to be allocated is: ( <code>pl[0].num_pts - 2</code> ) * <code>sizeof (Xgl_pt_f3d)</code>
---------------------	---

### ***Return Value***

Returns a 1 if the normal was computed successfully; otherwise, returns a 0.

## *XgliUtComputeTstarFn*

```
int XgliUtComputeTstarFn(  
    Xgl_geom_normal    geom_normal,  
    Xgl_boolean        normalize,  
    Xgl_pt_list*       point_list,  
    Xgl_pt_f3d*        facet_normal)
```

Computes the normal for the facets in the triangle star from the *point\_list* and returns the computed normals.

### ***Input Parameters***

<i>geom_normal</i>	Geometry normal format as defined by the API attribute XGL_3D_CTX_SURF_GEOM_NORMAL
<i>normalize</i>	If TRUE, normalizes the normal.
<i>point_list</i>	Data from which the normal is calculated.

### ***Output Parameter***

<i>facet_normal</i>	Caller allocated structure in which to return the computed normals. The memory to be allocated is: ( <i>pl</i> [0].num_pts - 2) * sizeof(Xgl_pt_f3d)
---------------------	---

### ***Return Value***

Returns a 1 if the normal was computed successfully; otherwise, returns a 0.

## *XgliUtComputeTstarFnPl*

```
int XgliUtComputeTstarFnPl(  
    Xgl_geom_normal    geom_normal,  
    Xgl_boolean        normalize,  
    Xgl_pt_list*       point_list,  
    Xgl_pt_list*       saved_pl,  
    Xgl_tlist_flags    tlist_flags,  
    Xgl_pt_f3d*        facet_normal)
```

Computes the normal for the facets in a triangle star (a subset of the triangle list primitive) from two input point lists provided by the user and returns the computed normals. Two input point lists are necessary in case the triangle star

vertex data is non-contiguous. The first vertex in the *saved\_pl* point list points to the first vertex in the triangle star. All other vertices in the triangle star are in *point\_list* beginning with the second location in *point\_list*.

#### **Input Parameters**

<i>geom_normal</i>	Geometry normal format as defined by the API attribute XGL_3D_CTX_SURF_GEOM_NORMAL.
<i>normalize</i>	If TRUE, normalizes the normal.
<i>point_list</i>	Vertex data for the second point through the last points in the triangle star.
<i>saved_pl</i>	Vertex data for the first point in the triangle star.
<i>tlist_flags</i>	Global triangle list flags that were passed into the <code>xgl_triangle_list()</code> primitive.

#### **Output Parameter**

<i>facet_normal</i>	Caller allocated structure in which to return the computed normals. The memory to be allocated is: ( <code>pl[0].num_pts - 2</code> ) * <code>sizeof (Xgl_pt_f3d)</code>
---------------------	---

#### **Return Value**

Returns a 1 if the normal was computed successfully; otherwise, returns a 0.

### ***XgliUtComputeVnReverse***

```
int XgliUtComputeVnReverse(
    Xgl_usgn32      num_pt_lists,
    Xgl_pt_list*    point_list,
    Xgl_pt_f3d*     vn)
```

This utility reverses the vertex normals in *point\_list*.

#### **Input Parameters**

<i>num_pt_lists</i>	Number of point lists in <i>point_list</i> .
<i>point_list</i>	Input from which the vertex normals should be reversed. .



**Output Parameter**

**vn** Caller allocated structure in which to return the reversed vertex normals. The memory to be allocated is:  
`tot_num_pts * sizeof(Xgl_pt_f3d)`  
where `tot_num_pts` is initially zero and for each point in the list `tot_num_pts += pl[i].num_pts`

**Return Value**

Returns a 1 if the function is successful; otherwise, returns a 0.

***XgliUtComputeZTolerance***

```
void XgliUtComputeZTolerance(  
    const Xgli_point_list*    pl,  
    float*                    z_offset)
```

Computes the *z\_offset* using the input point list *pl*. Used either when drawing edges or when the API attribute `XGL_3D_CTX_SURF_DC_OFFSET` is TRUE. The output value is added to the Z values of the points when drawing edges so that edges appear on top of the rendered surface. This value is also subtracted from the points of a surface primitive if the API attribute `XGL_3D_CTX_SURF_DC_OFFSET` is TRUE.

**Input Parameters**

**pl** Point list from which to calculate the Z offset.

**Output Parameter**

**z\_offset** The value of the computed Z offset.

***XgliUtCdDcCircleApprox***

```
Xgl_sgn32 XgliUtCdDcCircleApprox(  
    XglContext3d    *ctx,  
    XglViewGrp3dItf *viewGrpItf,  
    XglConicList3d  *circle_list)
```

Evaluates the number of points to be used to approximate a circle when the value of the attribute `XGL_CTX_NURBS_CURVE_APPROX` is `XGL_CURVE_METRIC_DC` or `XGL_CURVE_CHORDAL_DEVIATION_DC`.

**Input Parameters**

<i>ctx</i>	Pointer to a 3D Context.
<i>viewGrpItf</i>	Pointer to a 3D view group interface.
<i>circle_list</i>	Pointer to an XglConicList3d object containing a list of circles or circular arcs.

**Output Parameter**

None

**Return Value**

Returns the number of points to be used to approximate a circle.

***XgliUtDcCircleApprox***

```
Xgl_sgn32 XgliUtDcCircleApprox(
    XglContext3d      *ctx,
    XglViewGrp3dItf   *viewGrpItf,
    Xgl_circle_list   *circle_list)
```

Evaluates the number of points to be used to approximate a circle when the value of the attribute XGL\_CTX\_NURBS\_CURVE\_APPROX is XGL\_CURVE\_METRIC\_DC or XGL\_CURVE\_CHORDAL\_DEVIATION\_DC.

**Input Parameters**

<i>ctx</i>	Pointer to a 3D Context.
<i>viewGrpItf</i>	Pointer to a 3D view group interface.
<i>circle_list</i>	Pointer to a list of circles or circular arcs.

**Output Parameter**

None

**Return Value**

Returns the number of points to be used to approximate a circle.

## *XgliUtDcArcApprox*

```
Xgl_sgn32 XgliUtDcArcApprox(  
    XglContext3d      *ctx,  
    XglViewGrp3dItf  *viewGrpItf,  
    Xgl_arc_list      *arc_list)
```

Evaluates the number of points to be used to approximate an arc when the value of the attribute XGL\_CTX\_NURBS\_CURVE\_APPROX is XGL\_CURVE\_METRIC\_DC or XGL\_CURVE\_CHORDAL\_DEVIATION\_DC.

### ***Input Parameters***

<i>ctx</i>	Pointer to a 3D Context.
<i>viewGrpItf</i>	Pointer to a 3D view group interface.
<i>arc_list</i>	Pointer to a list of circular arcs.

### ***Output Parameter***

None

### ***Return Value***

Returns the number of points to be used to approximate an arc.

## *XgliUtCdDcEllArcApprox*

```
Xgl_sgn32 XgliUtCdDcEllArcApprox(  
    XglContext3d      *ctx,  
    XglViewGrp3dItf  *viewGrpItf,  
    XglConicList3d    *ell_list)
```

Evaluates the number of points to be used to approximate an ellipse when the value of the attribute XGL\_CTX\_NURBS\_CURVE\_APPROX is XGL\_CURVE\_METRIC\_DC or XGL\_CURVE\_CHORDAL\_DEVIATION\_DC.

### ***Input Parameters***

<i>ctx</i>	Pointer to a 3D context.
<i>viewGrpItf</i>	Pointer to a 3D view group interface.
<i>ell_list</i>	Pointer to an XglConicList3d object containing a list of elliptical arcs.

### **Output Parameter**

None

### **Return Value**

Returns the number of points to be used to approximate an ellipse.

## ***XgliUtDcEllArcApprox***

```
Xgl_sgn32 XgliUtDcEllArcApprox(
    XglContext3d      *ctx,
    XglViewGrp3dItf  *viewGrpItf,
    Xgl_ell_list      *ell_list)
```

Evaluates the number of points to be used to approximate an ellipse when the value of the attribute `XGL_CTX_NURBS_CURVE_APPROX` is `XGL_CURVE_METRIC_DC` or `XGL_CURVE_CHORDAL_DEVIATION_DC`.

### **Input Parameters**

<i>ctx</i>	Pointer to a 3D context.
<i>viewGrpItf</i>	Pointer to a 3D view group interface.
<i>ell_list</i>	Pointer to a list of elliptical arcs.

### **Output Parameter**

None

### **Return Value**

Returns the number of points to be used to approximate an ellipse.

## ***XgliUtFaceDistinguish***

```
const Xgli_surf_attr_3d* XgliUtFaceDistinguish(
    XglContext3d*      ctx,
    Xgl_pt_f3d*        normal,
    Xgl_pt_f3d*        pt,
    XglViewGrp3dItf*   view_itf)
```

Identifies a face to be either front-facing or back-facing.

**Input Parameters**

<i>ctx</i>	Context used in rendering the primitive.
<i>normal</i>	The facet normal of the face that is being distinguished.
<i>pt</i>	A point on the face that is being distinguished.
<i>view_itf</i>	The view group from which the eye vector is used in determining front versus back facing.

**Output Parameter**

None

**Return Value**

Returns either the front or back face attributes as a pointer to the *Xgli\_surf\_attr\_3d* structure.

***XgliUtGetZCompFunc***

```
void XgliUtGetZCompFunc(  
    Xgl_z_comp_method    method,  
    Xgl_boolean          (**func)(Xgl_usgn32, Xgl_usgn32))
```

Returns the Z-comparison function *func* based on the Z-comparison method.

**Input Parameter**

<i>method</i>	Z-comparison method for the function.
---------------	---------------------------------------

**Output Parameter**

<i>func</i>	The Z-comparison function.
-------------	----------------------------

## *XgliUtIsScreenDoor*

```
Xgl_boolean XgliUtIsScreenDoorTransparent(
    XglContext3d *    ctx,
    Xgl_boolean      front)
```

Determines whether a surface is screen door transparent; ignores the blending attributes.

### ***Input Parameters***

<i>ctx</i>	Context from which attributes are obtained.
<i>front</i>	If <i>front</i> is <code>TRUE</code> , the surface front attributes are checked; otherwise, the back attributes are checked.

### ***Output Parameter***

None

### ***Return Value***

Returns `TRUE` if the surface is transparent and `FALSE` otherwise.

## *XgliUtIsScreenDoorTransparent*

```
Xgl_boolean XgliUtIsScreenDoorTransparent(
    XglContext3d *    ctx,
    Xgl_boolean      front)
```

Determines whether the surface has blended transparency.

### ***Input Parameters***

<i>ctx</i>	Context from which attributes are obtained.
<i>front</i>	If <i>front</i> is <code>TRUE</code> , the surface front attributes are checked; otherwise, the back attributes are checked.

### ***Output Parameter***

None

### ***Return Value***

Returns `TRUE` if the surface is transparent and `FALSE` otherwise.

## *XgliUtIsTransparent*

```
Xgl_boolean XgliUtIsTransparent(  
    XglContext3d *    ctx,  
    Xgl_boolean      front)
```

Determines whether a surface is transparent.

### ***Input Parameters***

<i>ctx</i>	Context from which the attributes are obtained.
<i>front</i>	If <i>front</i> is TRUE, the surface front attributes are checked; otherwise, the back attributes are checked.

### ***Output Parameter***

None

### ***Return Value***

Returns TRUE if the surface is transparent and FALSE otherwise.

## *XgliUtIsTransparent*

```
Xgl_boolean XgliUtIsTransparent(  
    float            transparency,  
    Xgl_transp_method transp_method,  
    Xgl_blend_eq     blend_eq)
```

This function is similar to the other `XgliUtIsTransparent` utility except that it gets the API transparency attributes as arguments to the function.

### ***Input Parameters***

<i>transparency</i>	Value of the attribute <code>XGL_3D_CTX_FRONT TRANSP</code> or <code>XGL_3D_CTX_BACK TRANSP</code> . If face distinguishing is FALSE, then transparency is front.
<i>transp_method</i>	Value of <code>XGL_3D_CTX_SURF TRANSP METHOD</code> .
<i>blend_eq</i>	Value of <code>XGL_3D_CTX_SURF TRANSP BLEND EQ</code> .

### ***Output Parameter***

None

### **Return Value**

Returns `TRUE` if the surface is transparent and `FALSE` otherwise.

## ***XgliUtMeanWg***

```
void XgliUtMeanWg(
    Xgli_acolor*    vector,
    Xgl_usgn32      siz,
    float*          wg,
    Xgl_usgn32      num_channel,
    Xgli_acolor*    out_clr)
```

Accumulates the result of the product of the individual fields of *vector* array with the corresponding entries in the *wg* array for as many entries as given by *siz*. The number of channels in the *vector* array (and therefore in the *out\_clr*) is specified by *num\_channel*.

### **Input Parameters**

<i>vector</i>	The vector array that is weighted and accumulated.
<i>siz</i>	Number of entries in the vector array.
<i>wg</i>	The weights by which the vector array should be multiplied.
<i>num_channel</i>	Number of channels of useful information (maximum of 4) in the vector array.

### **Output Parameter**

<i>out_clr</i>	Output color.
----------------	---------------

## ***XgliUtMellaToPline***

```
Xgl_sgn32 XgliUtMellaToPline(
    XglContext3d    *ctx,
    XglViewGrp3dItf *viewGrpItf,
    Xgl_ell_list     *ell_list,
    Xgl_pt_list      **point_list,
    Xgl_facet_list   **facet_list)
```

Tessellates the 3D multielliptical arcs stored in *ell\_list*.



**Input Parameters**

<i>ctx</i>	Pointer to a 3D Context.
<i>viewGrpItf</i>	Pointer to a 3D view group interface.
<i>ell_list</i>	Pointer to a list of elliptical arcs.

**Output Parameters**

<i>point_list</i>	Point lists of the tessellated elliptical arcs. Any space that is required for the point lists is allocated by this routine.
<i>facet_list</i>	Facet list of the tessellated elliptical arcs. Any space that is required for the facet list is allocated by this routine. The value of <i>*facet_list</i> on return will always be NULL if XGL_CTX_ARC_FILL_STYLE is XGL_ARC_OPEN.

**Return Value**

Returns 1 if the elliptical arcs are successfully tessellated; otherwise, returns 0.

***XgliUtModelClipMarker***

```
Xgl_sgn32  XgliUtModelClipMarker(  
    XglContext3d*      ctx,  
    XglViewGrp3dItf*   view_grp,  
    Xgl_pt_list*        pl_in,  
    Xgl_pt_list**       pl_out)
```

Takes a single point list stored in *pl\_in* and model clips the points against the current model clipping planes. Note that only the center points of the markers are clipped – the individual marker shapes themselves are not.

**Input Parameters**

<i>ctx</i>	Context from which attributes are obtained.
<i>view_itf</i>	View group from which model clip planes are obtained.
<i>pl_in</i>	Input point list. This argument is the point list passed to <code>lilMultiPolyline()</code> .

**Output Parameter**

<i>pl_out</i>	List of points containing only those that are within the clipping planes.
---------------	---

### **Return Value**

Returns the number of points in the output list. It is the responsibility of the caller to free the memory that this routine allocates to hold the clipped points.

## ***XgliUtModelClipMpline***

```
Xgl_sgn32 XgliUtModelClipMpline(
    XglContext3d*      ctx,
    XglViewGrp3dItf*   view_itf,
    Xgl_usgn32          num_plines,
    Xgl_pt_list*        pl_in,
    Xgl_pt_list**       pl_out)
```

Model clips a list of polylines against the current model clipping planes.

### **Input Parameters**

<i>ctx</i>	Context from which attributes are obtained.
<i>view_itf</i>	View group from which model clip planes are obtained.
<i>num_plines</i>	Number of point lists in <i>pl_in</i> .
<i>pl_in</i>	Input point lists. This argument is the point list passed to <code>lilMultiPolyline()</code> .

### **Output Parameter**

<i>pl_out</i>	Clipped output polyline(s). Any space that is required for the polylines is allocated by this routine. It is the responsibility of the caller to free any memory allocated by this routine.
---------------	---

### **Return Value**

Returns the number of point lists in the output. A return value of 0 indicates that the entire multipolyline was trivially rejected.

## *XgliUtModelClipMspg*

```
Xgl_sgn32 XgliUtModelClipMspg(  
    XglContext3d*      ctx,  
    XglViewGrp3dItf*   view_grp,  
    Xgl_sgn32          num_pl,  
    Xgl_pt_list*        pl_in,  
    Xgl_pt_list**       pl_out,  
    Xgl_facet_list*     fl_in,  
    Xgl_facet_list**    fl_clipped)
```

This function is used to model clip lists of individual polygons, such as might be specified by a call to `xgl_multi_simple_polygon()`. The function handles multiple facets correctly, removing from the output list those that correspond to polygons that are trivially rejected.

### ***Input Parameters***

<i>ctx</i>	Context from which attributes are obtained.
<i>view_itf</i>	View group from which model clip planes are obtained.
<i>num_pl</i>	Number of point lists in <i>pl_in</i> .
<i>pl_in</i>	Input point lists. This argument is the point list passed to <code>lilMultiPolyline()</code> .
<i>fl_in</i>	Input facet list.

### ***Output Parameters***

<i>pl_out</i>	A list of point lists defining the clipped polygon.
<i>fl_clipped</i>	Facet list for the clipped polygon.

### ***Return Value***

Returns the number of clipped bounds. The number of output bounds is always less than or equal to the number of input bounds – extra point lists are not introduced. The caller must free any memory allocated by this routine.

## *XgliUtModelClipPgon*

```
Xgl_sgn32 XgliUtModelClipPgon(
    XglContext3d*      ctx,
    XglViewGrp3dItf*   view_grp,
    Xgl_sgn32          num_pl,
    Xgl_pl_list*        pl_in,
    Xgl_pl_list**       pl_out)
```

Model-clips an optionally multi-bounded polygon specified as a list of point lists in the *pl\_in* structure, against the current model clipping planes.

---

**Note** – This function is only appropriate for individual polygons. If more than one point list is passed in then it is assumed that the polygon is multi-bounded. Calling this routine with multi, separate bounded polygons may result in incorrect data.

---

### ***Input Parameters***

<i>ctx</i>	Context from which attributes are obtained.
<i>view_itf</i>	View group from which model clip planes are obtained.
<i>num_pl</i>	Number of point lists in <i>pl_in</i> .
<i>pl_in</i>	Input point lists. This argument is the point list passed to <code>lilMultiPolyline()</code> .

### ***Output Parameter***

<i>pl_out</i>	A list of point lists defining the clipped polygon.
---------------	---

### ***Return Value***

Returns the number of clipped bounds. The number of output bounds is always less than or equal to the number of input bounds – extra point lists are not introduced. The caller must free any memory allocated by this routine.

## *XgliUtModelClipPoint*

```
Xgl_boolean XgliUtModelClipPoint(  
    XglContext3d*      ctx,  
    XglViewGrp3dItf*   view_grp,  
    Xgl_pt_f3d*         pt)
```

Model clips the 3D model-coordinate point *pt* against the current model clipping planes specified by the Context and the view group interface object.

### ***Input Parameters***

<i>ctx</i>	Context from which attributes are obtained.
<i>view_grp</i>	View group from which model clip planes are obtained.
<i>pt</i>	Point to be model clipped.

### ***Output Parameter***

None

### ***Return Value***

If the point is determined to be inside the clipping planes, then the function returns TRUE, otherwise it returns FALSE.

## *XgliUtModelClipTstrip*

```
Xgl_sgn32 XgliUtModelClipTstrip(  
    XglContext3d*      ctx,  
    XglViewGrp3dItf*   view_grp,  
    Xgl_pt_list*        pl_in,  
    Xgl_facet_list*     fl_in,  
    Xgl_pt_list**       pl_out,  
    Xgl_facet_list**    fl_out)
```

Takes as input a single point list, and optionally a facet list, and model clips them against the current model clipping planes. The output is a list of point lists defining triangle strips that may have been created by model clipping the original input list. In practice there is usually only one such list as the clipper makes every attempt to keep everything together in one piece by introducing degenerate triangles where appropriate to link strips together. It is not

impossible, however, for there to be more than one list under some circumstances, so applications that use this utility are advised to assume that there can multiple output strips.

#### **Input Parameters**

<i>ctx</i>	Context from which attributes are obtained.
<i>view_itf</i>	View group from which model clip planes are obtained.
<i>pl_in</i>	Input point list. This argument is the point list passed to <code>lilMultiPolyline()</code> .
<i>fl_in</i>	Input facet list.

#### **Output Parameters**

<i>pl_out</i>	A list of point lists defining triangle strips that may have been created by model clipping the original input list.
<i>fl_out</i>	Facet list for <i>pl_out</i> .

#### **Return Value**

Returns the number of triangle strips in the clipped output. The caller must free any memory allocated by this routine.

### ***XgliUtPixRect48to32***

```
void XgliUtPixRect48to32(
    XglPixRectMem*    dst_buf,
    const Xgl_bounds_i2d* rect,
    const XglPixRectMem* src_buf,
    const Xgl_pt_i2d*   pos)
```

Copies a region of a 48-bit PixRect *src\_buf* to a 32-bit PixRect *dst\_buf*. The function is designed to do the copy-back of the accumulation buffer to an image buffer. *rect* applies to the source buffer and *pos* to the destination buffer.

#### **Input Parameters**

<i>src_buf</i>	The source buffer used in the copy operation. The source buffer should be a 48-bit pixrect.
<i>rect</i>	The rectangle in the source buffer that should be copied.

*pos* The position in the destination buffer to be used as starting position.

**Output Parameter**

*dst\_buf* The destination buffer in the copy operation. The destination buffer is a 32-bit PixRect.

## *XgliUtVertexFrontFacing*

```
Xgl_boolean XgliUtVertexFrontFacing(  
    Xgl_pt_f3d*      position,  
    Xgl_pt_f3d*      normal,  
    XglViewGrp3dItf* viewGrp,  
    Xgl_boolean      flipNormal)
```

Determines if a given vertex is front facing. See the XGL\_3D\_CTX\_SURF\_NORMAL\_FLIP man page.

**Input Parameters**

*position* Vertex position in MC.  
*normal* Normal at vertex  
*viewGrp* View group interface object  
*flipNormal* Value of the Context attribute XGL\_3D\_CTX\_SURF\_NORMAL\_FLIP.

**Output Parameter**

None

**Return Value**

Returns TRUE if the vertex is front facing and FALSE otherwise.

## *XgliUtVertexOrientation*

```
Xgl_boolean XgliUtVertexOrientation(
    Xgl_pt_list*      pl,
    XglViewGrp3dItf* viewGrp,
    Xgl_boolean*      vtxFrontFacing,
    Xgl_boolean      flipNormal)
```

Determines, for each point in a given point list, if it is front or back facing. It also determines whether the point list contains a silhouette edge. See the `XGL_3D_CTX_SURF_NORMAL_FLIP` man page.

### **Input Parameters**

<i>pl</i>	Point list
<i>viewGrp</i>	View group interface object
<i>flipNormal</i>	Context attribute <code>XGL_3D_CTX_SURF_NORMAL_FLIP</code>

### **Output Parameter**

<i>vtxFrontFacing</i>	For each array index <i>i</i> , TRUE if point <i>i</i> of the point list is front facing and FALSE if it is back facing.
-----------------------	--

### **Return Value**

Returns TRUE if the point list contains a silhouette edge and FALSE otherwise.

## *XgliUtCdVdcCircleApprox*

```
Xgl_sgn32 XgliUtCdVdcCircleApprox(
    XglContext3d      *ctx,
    XglViewGrp3dItf  *viewGrpItf,
    XglConicList3d    *circle_list)
```

Evaluates the number of points to be used to approximate a circle when the value of the attribute `XGL_CTX_NURBS_CURVE_APPROX` is `XGL_CURVE_METRIC_VDC` or `XGL_CURVE_CHORDAL_DEVIATION_VDC`.

### **Input Parameters**

<i>ctx</i>	Pointer to a 3D Context.
<i>viewGrpItf</i>	Pointer to a 3D view group interface.



*circle\_list* Pointer to an XglConicList3d object containing a list of circles or circular arcs.

**Output Parameter**

None

**Return Value**

Returns the number of points to be used to approximate a circle.

## *XgliUtVdcCircleApprox*

```
Xgl_sgn32 XgliUtVdcCircleApprox(  
    XglContext3d      *ctx,  
    XglViewGrp3dItf  *viewGrpItf,  
    Xgl_circle_list   *circle_list)
```

Evaluates the number of points to be used to approximate a circle when the value of the attribute XGL\_CTX\_NURBS\_CURVE\_APPROX is XGL\_CURVE\_METRIC\_VDC or XGL\_CURVE\_CHORDAL\_DEVIATION\_VDC.

**Input Parameters**

*ctx* Pointer to a 3D Context.  
*viewGrpItf* Pointer to a 3D view group interface.  
*circle\_list* Pointer to a list of circles.

**Output Parameter**

None

**Return Value**

Returns the number of points to be used to approximate a circle.

## *XgliUtVdcArcApprox*

```
Xgl_sgn32 XgliUtVdcArcApprox(
    XglContext3d      *ctx,
    XglViewGrp3dItf  *viewGrpItf,
    Xgl_arc_list      *arc_list)
```

Evaluates the number of points to be used to approximate an arc when the value of the attribute `XGL_CTX_NURBS_CURVE_APPROX` is `XGL_CURVE_METRIC_VDC` or `XGL_CURVE_CHORDAL_DEVIATION_VDC`.

### ***Input Parameters***

<i>ctx</i>	Pointer to a 3D Context.
<i>viewGrpItf</i>	Pointer to a 3D view group interface.
<i>arc_list</i>	Pointer to a list of circular arcs.

### ***Output Parameter***

None

### ***Return Value***

Returns the number of points to be used to approximate an arc.

## *XgliUtCdVdcEllArcApprox*

```
Xgl_sgn32 XgliUtCdVdcEllArcApprox(
    XglContext3d      *ctx,
    XglViewGrp3dItf  *viewGrpItf,
    XglConicList3d    *ell_list)
```

Evaluates the number of points to be used to approximate an ellipse when the value of the attribute `XGL_CTX_NURBS_CURVE_APPROX` is `XGL_CURVE_METRIC_VDC` or `XGL_CURVE_CHORDAL_DEVIATION_VDC`.

### ***Input Parameters***

<i>ctx</i>	Pointer to a 3D Context.
<i>viewGrpItf</i>	Pointer to a 3D view group interface.
<i>ell_list</i>	Pointer to an <code>XglConicList3d</code> object containing a list of elliptical arcs.

**Output Parameter**

None

**Return Value**

Returns the number of points to be used to approximate an ellipse.

***XgliUtVdcEllArcApprox***

```
Xgl_sgn32 XgliUtVdcEllArcApprox(  
    XglContext3d      *ctx,  
    XglViewGrp3dItf   *viewGrpItf,  
    Xgl_ell_list       *ell_list)
```

Evaluates the number of points to be used to approximate an ellipse when the value of the attribute XGL\_CTX\_NURBS\_CURVE\_APPROX is XGL\_CURVE\_METRIC\_VDC or XGL\_CURVE\_CHORDAL\_DEVIATION\_VDC.

**Input Parameters**

<i>ctx</i>	Pointer to a 3D Context.
<i>viewGrpItf</i>	Pointer to a 3D view group interface.
<i>ell_list</i>	Pointer to a list of elliptical arcs.

**Output Parameter**

None

**Return Value**

Returns the number of points to be used to approximate an ellipse.

***XgliUtCdWcCircleApprox***

```
Xgl_sgn32 XgliUtCdWcCircleApprox(  
    XglContext3d      *ctx,  
    XglViewGrp3dItf   *viewGrpItf,  
    XglConicList3d     *circle_list)
```

Evaluates the number of points to be used to approximate a circle when the value of the attribute XGL\_CTX\_NURBS\_CURVE\_APPROX is XGL\_CURVE\_METRIC\_WC or XGL\_CURVE\_CHORDAL\_DEVIATION\_WC.

**Input Parameters**

<i>ctx</i>	Pointer to a 3D context.
<i>viewGrpItf</i>	Pointer to a 3D view group interface.
<i>circle_list</i>	Pointer to an XglConicList3d object containing a list of circles or circular arcs.

**Output Parameter**

None

**Return Value**

Returns the number of points to be used to approximate a circle.

***XgliUtWcCircleApprox***

```
Xgl_sgn32 XgliUtWcCircleApprox(
    XglContext3d      *ctx,
    XglViewGrp3dItf  *viewGrpItf,
    Xgl_circle_list   *circle_list)
```

Evaluates the number of points to be used to approximate a circle when the value of the attribute XGL\_CTX\_NURBS\_CURVE\_APPROX is XGL\_CURVE\_METRIC\_WC or XGL\_CURVE\_CHORDAL\_DEVIATION\_WC.

**Input Parameters**

<i>ctx</i>	Pointer to a 3D context.
<i>viewGrpItf</i>	Pointer to a 3D view group interface.
<i>circle_list</i>	Pointer to a list of circles.

**Output Parameter**

None

**Return Value**

Returns the number of points to be used to approximate a circle.

## *XgliUtWcArcApprox*

```
Xgl_sgn32 XgliUtWcArcApprox(  
    XglContext3d      *ctx,  
    XglViewGrp3dItf  *viewGrpItf,  
    Xgl_arc_list      *arc_list)
```

Evaluates the number of points to be used to approximate an arc when the value of the attribute XGL\_CTX\_NURBS\_CURVE\_APPROX is XGL\_CURVE\_METRIC\_WC or XGL\_CURVE\_CHORDAL\_DEVIATION\_WC.

### ***Input Parameters***

<i>ctx</i>	Pointer to a 3D context.
<i>viewGrpItf</i>	Pointer to a 3D view group interface.
<i>arc_list</i>	Pointer to a list of circular arcs.

### ***Output Parameter***

None

### ***Return Value***

Returns the number of points to be used to approximate an arc.

## *XgliUtCdWcEllArcApprox*

```
Xgl_sgn32 XgliUtCdWcEllArcApprox(  
    XglContext3d      *ctx,  
    XglViewGrp3dItf  *viewGrpItf,  
    XglConicList3d    *ell_list)
```

Evaluates the number of points to be used to approximate an ellipse when the value of the attribute XGL\_CTX\_NURBS\_CURVE\_APPROX is XGL\_CURVE\_METRIC\_WC or XGL\_CURVE\_CHORDAL\_DEVIATION\_WC.

### ***Input Parameters***

<i>ctx</i>	Pointer to a 3D Context.
<i>viewGrpItf</i>	Pointer to a 3D view group interface.
<i>ell_list</i>	Pointer to an XglConicList3d object containing a list of elliptical arcs.

***Output Parameter***

None

***Return Value***

Returns the number of points to be used to approximate an ellipse.

***XgliUtWcEllArcApprox***

```
Xgl_sgn32 XgliUtWcEllArcApprox(
    XglContext3d      *ctx,
    XglViewGrp3dItf   *viewGrpItf,
    Xgl_ell_list       *ell_list)
```

Evaluates the number of points to be used to approximate an ellipse when the value of the attribute XGL\_CTX\_NURBS\_CURVE\_APPROX is XGL\_CURVE\_METRIC\_WC or XGL\_CURVE\_CHORDAL\_DEVIATION\_WC.

***Input Parameters***

<i>ctx</i>	Pointer to a 3D Context.
<i>viewGrpItf</i>	Pointer to a 3D view group interface.
<i>ell_list</i>	Pointer to a list of elliptical arcs.

***Output Parameter***

None

***Return Value***

Returns the number of points to be used to approximate an ellipse.

## Bounding Box Utilities

XGL utilities for checking for bounding boxes are in the file `CheckBbox.h`.

### *XgliUt2dCheckBbox*

```
Xgl_geom_status XgliUt2dCheckBbox(  
    XglContext2d*      ctx,  
    Xgl_primitive_type prim_type,  
    Xgl_bbox*          bbox,  
    XglViewGrp2dItf*   view_grp_itf)
```

Performs a bounding box check against the 2D clip volume and returns the geometry status of the bounding box. For more information, see the man page for `xgl_context_check_bbox()`.

#### **Input Parameters**

<i>ctx</i>	Pointer to a 2D Context.
<i>prim_type</i>	The bounding box primitive type.
<i>bbox</i>	Pointer to a bounding box which can be an <i>Xgl_bbox_i2d</i> , <i>Xgl_bbox_f2d</i> , or <i>Xgl_bbox_d2d</i> structure.
<i>view_grp_itf</i>	Pointer to a 2D view group interface.

#### **Output Parameter**

None

#### **Return Value**

This utility returns a geometry status which is either `XGL_GEOM_STATUS_VIEW_REJECT` (outside the clipping volume) or 0 (clipped).

## *XgliUt3dCheckBbox*

```
Xgl_geom_status XgliUt3dCheckBbox(
    XglContext3d*      ctx,
    Xgl_primitive_type prim_type,
    Xgl_bbox*          bbox,
    XglViewGrp3dItf*   view_grp_itf)
```

Performs a bounding box check against the 3D clip volume and returns the geometry status of the bounding box. For more information, see the man page for `xgl_context_check_bbox()`.

### **Input Parameters**

<i>ctx</i>	Pointer to a 3D Context.
<i>prim_type</i>	The bounding box primitive type.
<i>bbox</i>	Pointer to a bounding box which can be an <i>Xgl_bbox_f3d</i> or <i>Xgl_bbox_d3d</i> structure.
<i>view_grp_itf</i>	Pointer to a 3D view group interface.

### **Output Parameter**

None

### **Return Value**

Returns a geometry status which is a combination of the following flags:

```
XGL_GEOM_STATUS_VIEW_ACCEPT
XGL_GEOM_STATUS_VIEW_REJECT
XGL_GEOM_STATUS_VIEW_SMALL
XGL_GEOM_STATUS_MODEL_ACCEPT
XGL_GEOM_STATUS_MODEL_REJECT
```



## Copy Buffer Utilities

XGL utilities for copy buffer operations are in the file `CopyBuffer.h`.

### *XgliUtAdjustRectPos*

```
extern int XgliUtAdjustRectPos(  
    XglRaster*      src_dev,  
    Xgl_bounds_i2d* src_rect,  
    Xgl_bounds_i2d* adj_src_rect,  
    XglRaster*      dest_dev,  
    Xgl_pt_i2d*      dest_pos,  
    Xgl_pt_i2d*      adj_dest_pos);
```

Computes a new rectangle and position whose coordinates are valid as input to the `XgliUtCopyBuffer` utility. The new or adjusted rectangle and position are based upon the original rectangle and position and the size of the source and destination device.

#### **Input Parameters**

<i>src_dev</i>	Source device.
<i>src_rect</i>	Source rectangle in <i>src_dev</i> 's coordinate space.
<i>adj_src_rect</i>	Adjusted source rectangle; the new, valid rectangle.
<i>dest_dev</i>	Destination device.
<i>dest_pos</i>	Destination position in <i>dest_dev</i> 's coordinate space.
<i>adj_dest_pos</i>	Adjusted destination position.

#### **Output Parameter**

None

#### **Return Value**

Returns 1 if successful or 0 if the input data is inconsistent.

## *XgliUtCopyBuffer*

```
extern int XgliUtCopyBuffer(
    XglPixRect*          dest_pr,
    const Xgl_pt_i2d*    dest_pos,
    const Xgl_color*     dest_fg_color,
    const Xgl_color*     dest_bg_color,
    XglPixRectMem*       dest_clip_mask,
    XglPixRect*          src_pr,
    const Xgl_bounds_i2d* src_rect,
    XglPixRectMem*       src_clip_mask,
    Xgli_cb_color_info*   color_info,
    Xgli_cb_mask_and_rop_info* rop_info,
    Xgli_cb_fill_info*    fill_info,
    Xgli_cb_z_buffer_info* z_buffer_info)
```

Implements the `xgl_copy_buffer()` function. It copies a rectangular block of pixels from a source raster to a destination raster and, in addition, may convert from one color type to another, clip, fill with a pattern, or perform the copy based upon Z-buffer values. The caller must ensure that the device is locked (in other words, call `WIN_LOCK` if necessary).

### ***Input Parameters***

<i>dest_pr</i>	PixRect representing the destination raster.
<i>dest_pos</i>	Destination position.
<i>dest_fg_color</i>	Foreground color from destination Context.
<i>dest_bg_color</i>	Background color from destination Context.
<i>dest_clip_mask</i>	Per-pixel clip mask represented as a PixRect for the destination.
<i>src_pr</i>	PixRect representing the source raster.
<i>src_rect</i>	Rectangle of pixels which get copied to destination.
<i>src_clip_mask</i>	Per-pixel clip mask represented as a PixRect for the source.
<i>color_info</i>	Color info for source and destination raster; may be <code>NULL</code> if copy does not involve color.
<i>rop_info</i>	Plane mask and rop function; may be <code>NULL</code> if not doing ROP or masking.

<i>fill_info</i>	Information from destination Context for filling with patterns or stipples; may be NULL if not using patterned fill.
<i>z_buffer_info</i>	Information for copying from one Z-buffer to another; should be NULL if not copying Z-buffer data.

**Output Parameter**

None

**Return Value**

Returns 1 if the function succeeds or 0 for nonsuccess.

***XgliUtFbToMemCopyBuffer***

```
int XgliUtFbToMemCopyBuffer(  
    XglContext*      dest_ctx,  
    Xgl_bounds_i2d*  rect,  
    Xgl_pt_i2d*      pos,  
    XglRaster*       src_dev,  
    XglPixRect*      src_image_pr,  
    XglPixRect*      src_zbuffer_pr,  
    XglPixRectMem*   src_clip_mask);
```

A high-level utility that implements `xgl_copy_buffer()` when copying from a Device's frame buffer to a Memory Raster. This function calls `XgliUtCopyBuffer()`. The caller must ensure that the device is locked (in other words, `WIN_LOCK` is called if needed).

**Input Parameters**

<i>dest_ctx</i>	Destination Context object
<i>rect</i>	Rectangle from API
<i>pos</i>	Position from API
<i>src_dev</i>	Source device
<i>src_image_pr</i>	PixRect for source device's image buffer.
<i>src_zbuffer_pr</i>	PixRect for source device's Z buffer
<i>src_clip_mask</i>	PixRect for source device's clip mask

## ***Output Parameter***

None

## ***Return Value***

Returns 1 if the function succeeds or 0 for nonsuccess.

## ***XgliUtGetMaskAndRopFunc***

```
extern void XgliUtGetMaskAndRopFunc(  
    Xgl_rop_mode      rop,  
    Xgl_usgn32 (**func)(Xgl_usgn32 s, Xgl_usgn32 d, Xgl_usgn32 p))
```

Returns a pointer to a function that implements the given ROP value.

## ***Input Parameter***

*rop*                      ROP mode that is implemented by the returned function.

## ***Output Parameter***

*func*                    Pointer to a function which returns the ROP'ed and masked pixel; it takes the following parameters: the source pixel *s*, the destination pixel, *d*, and the plane mask *p*.

## Polygon Classification Utilities

XGL utilities for polygon classification are in the file `PgonClass.h`.

### *XgliUtClassifyMsp*

```
int XgliUtClassifyMsp(
    Xgl_usgn32          num_pt_lists
    Xgl_pt_list*        point_list
    Xgl_pt_f3d*         points
    Xgli_polygon_class* pgon_class
```

Classifies polygons sent to the primitive `xgl_multi_simple_polygon()`. The classification consists of checking the number of points in each polygon and testing for convexity. The information obtained can be used to decrease rendering time. For example, if a classified polygon has the `XGL_PGON_TRISTAR` bit set then a triangle star renderer can be used rather than a generic polygon scan converter.

#### **Input Parameters**

<i>num_pt_lists</i>	Number of point lists in <i>point_list</i> .
<i>point_list</i>	Pointer to the data to be classified.
<i>points</i>	Pointer to the polygon's facet normal list. This list must be of type <code>XGL_FACET_NORMAL</code> or <code>XGL_FACET_COLOR_NORMAL</code> . If calling with a 2D point list, this parameter is ignored.

#### **Output Parameter**

<i>pgon_class</i>	A pointer to a bit vector. The bit vector must be allocated in the calling routine to <i>num_pt_lists</i> in size. The bit vector array is returned with at least one of the bits set. See the return value for <code>XgliUtClassifyPgon</code> for the possible values.
-------------------	--

#### **Return Value**

Returns 0 if the classification finished successfully and 1 if the classification was aborted. An attempt is made to classify a 3D multisimple polygon list without the facet normal list.

## *XgliUtClassifyPgon*

```
Xgli_polygon_class XgliUtClassifyPgon(
    Xgl_usgn32          num_pt_lists
    Xgl_pt_list*        pl
    Xgl_pt_f3d*         facet_normal)
```

Classifies polygons sent to the primitive `xgl_polygon()`. The classification consists of checking the number of points in the polygon, checking the number of bounds, and testing for convexity. The information obtained can be used to decrease rendering time. For example, if a classified polygon has the `XGL_PGON_TRISTAR` bit set, a triangle star renderer can be used rather than a generic polygon scan converter.

### ***Input Parameters***

<i>num_pt_lists</i>	Number of point lists in <i>pl</i> .
<i>pl</i>	Pointer to the data to be classified.
<i>facet_normal</i>	Pointer to the polygon's facet normal. This must point to a facet of type <i>Xgl_normal_facet</i> or <i>Xgl_color_normal_facet</i> . If calling with a 2D point list, this parameter is ignored.

### ***Output Parameter***

None

### ***Return Value***

Returns a bit vector with at least one of the following set:

- `XGL_PGON_DEGENERATE` – The polygon has less than three points in its point list.
- `XGL_PGON_SIDES_ARE_3` – The polygon has three points in its point list. No testing is done for degenerate data.
- `XGL_PGON_SIDES_ARE_4` – The polygon has four points in its point list. No testing is done for degenerate or self-intersecting data.
- `XGL_PGON_SIDES_UNSPECIFIED` – The polygon has more than four points in its point list. No testing is done for degenerate data nor for a self intersecting point list.

- XGL\_PGON\_TRISTAR – The polygon can be rendered using a triangle star starting from the first vertex in the point list.
- XGL\_PGON\_CONV\_ONEBOUND – The polygon is convex (can be rendered as a triangle star from any vertex) and is single bounded (there are no holes in the polygon).
- XGL\_PGON\_COMPLEX – No information was found about the polygon.

## Polygon Decomposition Utilities

XGL provides utilities to decompose polygons into triangles in the file `utils.h`.

### *XgliUtDecomposePgon*

```
int XgliUtDecomposePgon(  
    Xgl_facet_type      facet_type,  
    Xgl_facet*          facet,  
    Xgl_usgn32          num_in_pt_lists,  
    Xgl_pt_list*        in_pl,  
    Xgl_usgn32*         num_out_pt_lists,  
    Xgl_pt_list**       out_pl,  
    Xgl_color_type      color_type,  
    Xgl_pt_f3d*         d_c_normal,  
    Xgl_geom_normal     geom_normal_type,  
    Xgl_boolean         normal_flip);
```

Decomposes one complex polygon facet into strips of triangle stars, which are returned via the output parameter *out\_pl*. The utility allocates the memory for the output point lists of triangle stars, so it's the caller's responsibility to free the memory when the output point lists are no longer needed.

#### **Input Parameters**

<i>facet_type</i>	Facet type of input polygon
<i>facet</i>	Facet information
<i>num_in_pt_lists</i>	Number of point lists (i.e. bounds) in input polygon.
<i>in_pl</i>	Array of point lists for input polygon.
<i>color_type</i>	Color type (index or RGB).
<i>d_c_normal</i>	Normalized facet normal of input polygon.

<i>geom_normal_type</i>	Geometry normal format as defined by the API attribute XGL_3D_CTX_SURF_GEOM_NORMAL.
<i>normal_flip</i>	Specifies whether vertex and facet normals are flipped, as defined by the attribute XGL_3D_CTX_SURF_NORMAL_FLIP.

### **Output Parameters**

<i>num_out_pt_lists</i>	Number of output point lists of triangle stars.
<i>out_pl</i>	Pointer to output point lists of triangle stars.

### **Return Value**

Returns 1 if the polygon is successfully decomposed and 0 if memory allocation fails.

## ***XgliUtDecomposeNsiPgon***

```
int XgliUtDecomposeNsiPgon(
    Xgl_facet_type      facet_type,
    Xgl_facet*          facet,
    Xgl_usgn32           num_in_pt_lists,
    Xgl_pt_list*         in_pl,
    Xgl_usgn32*          num_out_pt_lists,
    Xgl_pt_list**        out_pl,
    Xgl_color_type       color_type,
    Xgl_pt_f3d*          d_c_normal,
    Xgl_geom_normal      geom_normal_type,
    Xgl_boolean          normal_flip);
```

Decomposes one non-self-intersecting polygon facet into strips of triangle stars, which are returned via the output parameter *out\_pl*. The utility allocates the memory for the output point lists of triangle stars, so it's the caller's responsibility to free the memory when the output point lists are no longer needed.

### **Input Parameters**

<i>facet_type</i>	Facet type of input polygon.
<i>facet</i>	Facet information.
<i>num_in_pt_lists</i>	Number of point lists (i.e. bounds) in input polygon.



---

<i>in_pl</i>	Array of point lists for input polygon.
<i>color_type</i>	Color type (index or RGB).
<i>d_c_normal</i>	Normalized facet normal of input polygon.
<i>geom_normal_type</i>	Geometry normal format as defined by the API attribute XGL_3D_CTX_SURF_GEOM_NORMAL.
<i>normal_flip</i>	Specifies whether vertex and facet normals are flipped, as defined by the API attribute XGL_3D_CTX_SURF_NORMAL_FLIP.

**Output Parameters**

<i>num_out_pt_lists</i>	Number of output point lists of triangle stars.
<i>out_pl</i>	Pointer to output point lists of triangle stars.

**Return Value**

Returns 1 if the polygon is successfully decomposed and 0 if memory allocation fails.



## *Performance Tuning*

---



This appendix presents information about performance tuning. Tuning code for performance can be broken down into two distinct parts: finding the performance critical paths and tuning those paths.

This appendix details methodologies for finding performance hot spots and describes both high-level and low-level techniques for alleviating them. The following topics are covered:

- Finding the performance critical paths
- Selecting good benchmarks
- Tuning the performance critical paths
- Tips and techniques for faster code

## Finding the Performance Critical Paths

Being able to find the performance critical paths is as important as tuning them. However, finding these paths is not always easy. Your intuition about where the performance problems lie can mislead you. Unless you are personally familiar with a particular section of code, it is best to approach this process with no preconceptions and to gather profile information from an application to direct your investigation.

There are currently three ways you can gather profile information. These methods are introduced here and described more fully in “Tuning Performance Critical Paths” on page 415.

### 1. Build profile libraries.

Libraries built with the `-pg` option produce `gprof` output. This output gives you a very close approximation of how much time was spent in each function of the library, an exact count of how many times each function was executed, and a function call graph.

The disadvantages of profile libraries are that they must be compiled using special flags, they must be built statically (this restriction may be removed at a later time), they don’t measure system time, they require re-linking the application with `-pg`, and they don’t provide any information about the memory system (for example, page faults).

Although profile libraries have disadvantages, they are currently the only standard mechanism capable of providing function call counts and the function call graph. These capabilities make profile libraries a very attractive analysis tool for in-depth performance tuning.

### 2. Use the performance collector and analyzer tools included *ProWorks*.

The *collector* is used from the debugger to gather information about a program while it is running, and the *analyzer* is a user interface that sorts and displays that information in various ways.

This tool has the benefit of being able to measure any code that hasn’t been stripped. No special compile flags are needed, and it doesn’t matter whether libraries are dynamically or statically linked or even `dlopen`’d. It also is aware of page faults.

One attractive feature of the analyzer is that it shows you the total amount of time spent in each shared library. This is useful for doing a overall analysis of your program. For example, if you're spending a lot of time in *libc*, you are probably doing a lot of `malloc`, `free`, or signal handling. Even though the analyzer can't show you what routine is calling these routines, you as a library developer may immediately know where to start looking.

However, this tool is not able to show the function call graphs or counts on the number of times a function was executed.

### 3. Use the Shared Library Interposer.

The Shared Library Interposer (SLI) installs hooks to trap function calls, which is a Sun OS 5.x special feature. However, it can't catch C++ virtual functions or static functions. The SLI can work on any number of shared libraries at the same time.

A disadvantage of SLI is that it requires additional interposing libraries to work. If you're just interested in measuring the performance of your API without the details of the code underneath, then these interposing libraries can be constructed once and easily be referenced later on. If you want the details of all the internal functions that were called, then you need to point SLI at your source tree and construct a new interposing library any time functions are created, destroyed or renamed (but not if just the body of functions were changed). This rebuilding takes approximately ten minutes for large libraries.

Unlike all of the above options, SLI gets *exact* time for each function. It does this by bracketing each function call with `gethrtime()`. All the other schemes interrupt the process every 10 milliseconds or so and note which function they are in. Therefore, all of the non-SLI schemes only generate statistical approximations to how much time was spent in each function. Assuming your application runs for at least a few seconds, this statistical approximation is quite close to reality. SLI does not work with static libraries or applications, nor does it know about the memory system.

SLI has a GUI to allow easy interpretation of the gathered data. SLI also has the ability to log all the API calls and their arguments made during a session for later playback. The functionality has to be coded into SLI; therefore, it only works on a small number of libraries.

## At-a-Glance Comparison of Performance Tools

Table A-1 compares the different performance tools used to gather profile information.

*Table A-1* Comparing Applications Used to Gather Profile Information

Features	-pg	Collector	SLI
Time spent in each function	Y	Y	Y
Call counts for each function	Y		Y
Function call graph	Y		Y
Measures system time		Y	
Page fault aware		Y	
Library can be dynamic		Y	Y
Library can be static	Y	Y	
Handles multiple libraries	Y	Y	Y
Does not need special compile flags		Y	Y
No recompilation/linking of application needed		Y	Y
Works on <code>dlopen()</code> 'd libraries		Y	Y
Has a GUI for display		Y	Y
Needs no additional libraries	Y	Y	
Internal library functions measured	Y	Y	Y <sup>1</sup>
Virtual/static functions profiled	Y	Y	
Supports playback of library calls			Y <sup>2</sup>

1. If functions are created/destroyed/renamed, then the new interposing libraries need to be created for SLI. This does not take a tremendous amount of time, but it is an additional step.

2. Only a handful of libraries support this feature.

## *Recommendations for Performance Tools*

Choosing a performance analysis method is a matter of individual preference. This section provides recommendations, and you can determine which methods you are most comfortable with.

If you are interested in giving one or two areas a boost in performance, but the areas are not critical, use the collector and see if it can give you the information you need. If you're going to be spending a lot of time tuning code or if the collector does not meet your needs, then it's worth the effort to build a profile library.

SLI is not recommended for library developers because SLI's strength is in logging the library API calls for statistical analysis of how the library is used. As such, it is more useful for tuning the application to use the library more efficiently than it is for tuning the library itself.

For serious performance tuning, the profile library is recommended over the collector tool. This is because the collector does not produce the function call graphs or function call counts which are crucial for finding and tuning your critical paths.

All of the above schemes work at a functional level. If you are interested in finding out how many times a line of code is executed within a function, see the `tcov` man pages.

## *Selecting Good Benchmarks*

When searching for performance bottlenecks, it is important to use the right benchmarks. It is often easy to find and fix a bottleneck that makes benchmarks run faster, while the performance of real customer applications remains unchanged. In an ideal world, all performance tuning would be guided by real customer workload. In our less than ideal world, we must use approximations to real customer workload.

It is becoming more and more of a requirement for vendors to run a customer's application as part of the sales process. Improving your marketing-oriented benchmarks undoubtedly catches the customer's eye. However, improving your customer-oriented benchmarks will help close the sale and generate future business from your satisfied customers. In today's market, both types of

benchmarks need to be used to maximize your company's profit. The advantages and disadvantages of the three types of benchmarks are discussed below.

- Shared Library Interposer – Customer oriented

The Shared Library Interposer (SLI) is an excellent tool for logging and playing back library calls from real customer applications. Not only will SLI help you find exactly what needs to be tuned for a given application, it will allow you to give feedback to the application writers on how to use your library more effectively. Unfortunately, there is currently no industry standard way of reporting performance of real customer applications.

- Raw primitive benchmarks – Marketing oriented

Be cautious about using raw primitive benchmarks to guide tuning efforts. Although these benchmarks are good tools to measure peak performance, they produce results that are the least likely to match real customer workloads. However, reporting peak performance numbers is still the most common way for vendors to market their products. These benchmarks are well-suited for tuning the inner loops of a particular primitive's rendering code, and they can help identify library overhead for poorly-batched primitives (like single vector polylines).

- GPC Picture Level Benchmark – Customer and marketing oriented

The GPC Picture Level Benchmark's (PLB) exploits the strengths of real customer applications and raw primitive benchmarks. It is currently the closest industry standard benchmark to real customer applications. Because it's a standard, it allows you to compare your product against the competition. Improved results will translate well to customer-visible performance improvements.



## *Tuning Performance Critical Paths*

Performance tuning can be considered as occurring on three different levels. The first level involves looking for a central body of code, in which the application spends most of its time. The second level of performance tuning consists of algorithmic improvement, and the third level involves tuning assembly language.

### *Locating the Central Body of Code*

The first level of performance tuning involves looking at your profile output and checking for any obvious problems. For example, there might be a transform evaluation on every primitive, or `new` and `delete` might be called for every primitive. Fixing these types of problems usually requires little work. A simple yet extremely useful performance technique for this is to cache values in software for later use. You may also need to add an `if` test in several places and restructure the code, but the basic algorithm can remain intact. The difficult part in fixing these bugs is finding them. If a lot of time is spent in some functions that you know shouldn't be called frequently, then the collector will point you to the problem. If this isn't the case, then you may need to use the profile libraries so that you get the count and call graph information. Finally, `gprof` output may show you that you are calling a function many more times than you expected.

### *Changing the Underlying Algorithm*

The next level of performance tuning involves changing the underlying algorithm. Some examples of this are speeding up the special cases of a general algorithm, using fixed point arithmetic instead of floating point, using software caching schemes, and reducing the number of `malloc/free` calls from many to one. This type of tuning is frequently needed when a feature of the library that was previously deemed unimportant turns out to be useful to customers. Because the feature had a low priority, not much time was originally spent implementing it efficiently. Now it's necessary to go back and perform an in-depth analysis to make it run fast.

Coming up with a new algorithm requires designers to have a clear understanding of what is fast and what is slow on the current hardware. Some performance techniques are widely known (square root is slower than addition, hash tables speed up linked list searches, multiplication is faster than

division, and so on), but there are dozens of techniques that you can use to make algorithms more efficient. See “Tips and Techniques for Faster Code” for a discussion of these techniques.

## *Tuning at the Assembly Language Level*

To really tune a chunk of code well, you must look at the assembly language output of the compiler<sup>1</sup>. At some point, the algorithm you’re working on must be turned into machine-readable format. You need to ensure that all the effort you’ve put into avoiding expensive operations isn’t being overshadowed by some unintelligent process the compiler is doing. This type of tuning should be reserved for your performance-critical paths. Spending time tuning your code to produce near-optimal assembly output isn’t free. Reality dictates that you spend your time using the most cost-effective philosophy. It is good practice to frequently look at the assembly output. As you gain experience, looking at this information can become part of your development process, and it becomes a good way to verify your design.

## *Tips and Techniques for Faster Code*

As you read through the suggested techniques in this section, you will note that a knowledge of assembly language can be useful. Many of the techniques presented here can be applied without inspecting the assembly output, but a knowledge of assembly language becomes more essential as you progress through the tuning techniques suggested here.

## *Tune the Innermost Loops First*

Once you have identified your performance-critical paths, you need a starting point to begin your tuning. The way to get the most cost-effective performance is to tune the innermost loops first. These loops are executed many times (potentially hundreds of times) for every iteration of an outer loop. Once the

---

1. There is a big difference between reading the assembly output from the compiler and writing assembly language routines. It’s very similar to being able to read a foreign language versus being able to speak it. If you’re not totally familiar with a particular instruction, you can make an educated guess at what it is or look it up in a manual. This is much easier than creating an assembly language routine from scratch.

absolute innermost loop has been tuned, start expanding your view outward to the next innermost loops. Time permitting, continue out to your library entry points.

By tuning your innermost loops first, you can substantially increase the performance of your moderately and highly batched cases. The performance of your poorly batched cases may improve slightly, but not by much. The performance of poorly batched cases tend to be limited by start up costs and has little to do with the inner loops. Improving the performance of poorly batched cases is a more difficult task than tuning highly batched performance, and it requires applying virtually all of the tips in this appendix. If you are interested in tuning for poorly batched cases, assume that every loop is executed once and start counting CPU cycles in your assembler output. You will need to look all the way from your library entry point down to the lowest level function.

A simple source code transformation to improve inner loops is moving loop invariant code outward. Suppose you want to construct the vertices of a sphere. The straightforward implementation of this is as follows:

```
for ( theta=0.0 ; theta<2.0*PI ; theta+=theta_step ) {  
    for ( omega=-0.5*PI ; omega<0.5*PI ; omega+=omega_step ) {  
        pt->x = cos(theta)*cos(omega);  
        pt->y = sin(theta)*cos(omega);  
        pt->z = sin(omega);  
        pt++;  
    }  
}
```

This could easily be changed to:

```
for ( theta=0.0 ; theta<2.0*PI ; theta+=theta_step ) {  
    cos_theta = cos(theta);  
    sin_theta = sin(theta);  
    for ( omega=-0.5*PI ; omega<0.5*PI ; omega+=omega_step ) {  
        cos_omega = cos(omega);  
        pt->x = cos_theta*cos_omega;  
        pt->y = sin_theta*cos_omega;  
        pt->z = sin(omega);  
        pt++;  
    }  
}
```

You have reduced the number of calls to *cos()* and *sin()* in our inner loop from 5 to 2. Assuming that *omega\_step* is small enough that the inner loop executes a large number of times relative to the outer loop, you should see a performance increase by a factor of 2.5. You could try exploiting the relationship  $\cos^2 + \sin^2 = 1$ , but that would depend on the relative speeds of *cos/sin* and *sqrt*. If the hardware supports *sqrt*, use it.

The above example was a fairly simple one. Less obvious cases are more prevalent. Below is an example for copying one string to another. Assume the string structure has a *length* field and a character array with sufficient space to hold the string:

```
dest->length = src->length;
for ( i=0 ; i<src->length ; i++ ) {
    dest->string[i] = src->string[i];
}
```

Since you are operating through pointers, the compiler cannot assume that *src->length* isn't changed during each iteration of the loop. To get around this, keep the string length in a local variable as shown below:

```
length = dest->length = src->length;
for ( ; length>=0 ; length-- ) {
    dest->string[length] = src->string[length];
}
```

On SPARC, the upper loop takes six instructions per iteration while the bottom loop takes four instructions. This simple example shows us the most important lesson that can be learned about performance tuning and that is, don't trust the compiler. No matter how efficient the compiler gets, it cannot surpass a knowledgeable programmer.

As you move loop invariant code outward, you'll notice a proliferation of local variables. This is perfectly acceptable. These local variables can be thought of as a cache of values created by the programmer. While there are cases where too many local variables hurt performance, they are rare and their penalties are low in comparison to the much more likely gains they offer. It is quite common for local variables to map directly to hardware registers and never get stored to memory. One way to help the compiler to realize this is to declare local variables within the smallest scope they will be used in.

## *Don't Optimize Uncommon Cases at the Expense of Common Cases*

Although this rule is intuitively obvious, it is perhaps the easiest to forget. It is often quite tempting to add code that makes a seldom used operation run faster. At times you will need to add a little logic someplace else to make this optimization work. Note how this affects your common cases, and if it does, make sure that the performance trade-offs you are making are good ones.

This rule could also be called a “keep it simple” rule. To a first approximation, the more complicated and convoluted the code, the slower it will run. If you're just getting started in performance tuning, then go for simplicity. After you've had a chance to get familiar with the types of trade-offs that are made in the name of performance, you'll be in a better position to estimate the consequences of additional complexity.

## *Special-Case the Common Cases*

Most libraries have a set of attributes that can be changed by the user. Libraries will need to switch on the attribute or employ a hierarchical set of `if` tests to decode the attribute. In either case, it can be worthwhile to special case a small number of the most frequently set attributes (like line color). By having an `if` test that succeeds most of the time, you can decrease the average amount of time spent setting attributes for most applications. Certainly an application that never sets the line color will run slower, but the difference is likely to be quite small since attributes will have to go through the full decode cycle.

## *Choose Your Software Layers Carefully*

You need to define software layers that don't limit performance. An example is when `A` calls `B(X)`. The function `A` knows some property of `X` which `B` does not, so `B` spends time checking for the property, or it simply does things more generally (and less efficiently). Either `A` and `B` should be in the same software layer, or perhaps a special case version of `B` can be written which assumes the knowledge of the properties for `X`.

An example is `memcpy()`, which assumes only character-aligned data. If you are copying word-aligned data (or double-word aligned data), you can copy faster than `memcpy()` with a simple loop.

## *Move If Tests Outward*

Although `if` tests are certainly necessary for programming, it is advantageous to remove as many of them as possible from your performance critical paths. In today's high clock rate, super-scalar RISC chips, each branch in the code carries along with it the possibility of dozens of wasted CPU cycles.

Removing `if` tests can often mean replicating code. A simple example of this is shown below for drawing a polyline on a hardware device where the first vertex must be handled differently from all subsequent vertices:

```
first_vertex = 1;
for ( i=0 ; i<num_pts ; i++ ) {
    vertex_registers[X_OFFSET] = pt->x;
    vertex_registers[Y_OFFSET] = pt->y;
    vertex_registers[Z_OFFSET] = pt->z;
    pt++;
    if (!first_vertex) {
        // wait for DRAW operation to finish
        while(vertex_registers[DRAW_STATUS] != ALL_DONE) ;
    }
    first_vertex = 0;
}
```

This code can be restructured as:

```
vertex_registers[X_OFFSET] = pt->x;
vertex_registers[Y_OFFSET] = pt->y;
vertex_registers[Z_OFFSET] = pt->z;
for ( i=1 ; i<num_pts ; i++ ) {
    pt++;
    vertex_registers[X_OFFSET] = pt->x;
    vertex_registers[Y_OFFSET] = pt->y;
    vertex_registers[Z_OFFSET] = pt->z;
    // wait for DRAW operation to finish
    while(vertex_registers[DRAW_STATUS] != ALL_DONE) ;
}
```

By replicating the code which sends the vertex information to the hardware, an `if` test on every vertex was removed.

The above example was a simple one because it dealt with a very small and manageable portion of code. As you expand your focus outward from the innermost loops, it gets more and more difficult to replicate code. You start to

have huge chunks of code, each of which does basically the same thing. This becomes a maintenance problem. One technique for overcoming this is to have source code files with `#ifdefs` that are included by other files. Suppose you wanted to augment the above line renderer to handle polylines with color at each vertex. The straightforward way to do this is with an `if` test inside the inner loop:

```
vertex_registers[X_OFFSET] = pt->x;
vertex_registers[Y_OFFSET] = pt->y;
vertex_registers[Z_OFFSET] = pt->z;
for ( i=1 ; i<num_pts ; i++ ) {
    pt++;
    vertex_registers[X_OFFSET] = pt->x;
    vertex_registers[Y_OFFSET] = pt->y;
    vertex_registers[Z_OFFSET] = pt->z;
    if (pt_type & VERTEX_COLOR) {
        vertex_registers[R_OFFSET] = pt->color.r;
        vertex_registers[G_OFFSET] = pt->color.g;
        vertex_registers[B_OFFSET] = pt->color.b;
    }
    // wait for DRAW operation to finish
    while(vertex_registers[DRAW_STATUS] != ALL_DONE);
}
```

This keeps your maintenance costs down but at the expense of performance. If things like line patterning, homogeneous coordinates, or vertex flags are added, you will end up with a large number of `if` tests performed for every vertex. Fortunately, you can keep the maintenance costs down and still have optimal performance by keeping the code below in a separate file called `PolylinesProto.h`:

```
{
    // setup code here (probably with #ifdef's in it)

    vertex_registers[X_OFFSET] = pt->x;
    vertex_registers[Y_OFFSET] = pt->y;
    vertex_registers[Z_OFFSET] = pt->z;
    for ( i=1 ; i<num_pts ; i++ ) {
        pt++;
        vertex_registers[X_OFFSET] = pt->x;
        vertex_registers[Y_OFFSET] = pt->y;
        vertex_registers[Z_OFFSET] = pt->z;
    #    ifdef(VERTEX_COLOR)
        vertex_registers[R_OFFSET] = pt->color.r;
        vertex_registers[G_OFFSET] = pt->color.g;
        vertex_registers[B_OFFSET] = pt->color.b;
    #    endif
    // wait for DRAW operation to finish
    while(vertex_registers[DRAW_STATUS] != ALL_DONE) ;
    }
}
```

The PolylinesProto.h file is included in another file as shown below:

```
#define FUNCNAME PolylinesXyz
#undef VERTEX_COLOR
#include PolylinesProto.h
#undef FUNCNAME

#define FUNCNAME PolylinesXyzRgb
#define VERTEX_COLOR
#include PolylinesProto.h
#undef FUNCNAME
```

Thus, whenever a bug is filed, you fix it once for all polyline renderers. Unlike embedding `if(constant_expression)`'s in a macro definition, this technique allows the debugger to step through the `#include'd` code. This technique was used for the Sun XGL GX pipeline, which has nearly one hundred special purpose polyline renderers.



## *Unroll Loops Where Appropriate*

It was shown above that it is worthwhile to minimize the number of `if` tests on your performance critical paths. This rule applies to loops as well. If you have some knowledge about how your loop is going to be used, then you can exploit that knowledge to reduce the number of branches in your code along with your loop overhead. Just like the examples in the preceding section, this means replicating code. Unlike the preceding section, loop unrolling is only effective inside loop constructs and, therefore, is really only applicable in your innermost loops.

One example of where this can be used is in an `xgl_multi_simple_polygon()` rendering routine. You could have a specialized renderer which handles the case where the `SIDES_ARE_3` flag is set. Instead of having an outer loop for each polygon and an inner loop for each vertex, the inner loop can be completely unrolled to send down three vertices at a time. This way, you have reduced the number of `if` tests per polygon from four to one and saved other per loop-iteration overhead such as incrementing loop variables. A similar optimization could be used for `SIDES_ARE_4` (possibly be used in conjunction with `XGL_FACET_FLAG_SHAPE_CONVEX`).

Another common area for loop unrolling is in memory copy operations. The canonical copy operation shown below takes six instructions to copy each word of data when compiled at `-O2` on SPARC (`-O4` does some loop unrolling for you). Of these six instructions, only two are actually useful (the loading of the `src` value and the storing of that value to `dst`). The other four instructions are purely loop overhead (testing `size`, incrementing `dst`, incrementing `src`, and decreasing `size`).

```
for ( ; size>0 ; size-- ) {  
    *dst++ = *src++;  
}
```

By unrolling the loop once, you can get the loop to use nine instructions to copy two words of data. The example of unrolling the loop once is as follows:

```
for ( ; size>1 ; size-=2) {
    dst[0] = src[0];
    dst[1] = src[1];
    dst += 2;
    src += 2;
}
if (size) {
    *dst = *src; // in case "size" is odd
}
```

Unrolling the loop again produces 13 instructions to handle four words of data. Remembering that two instructions per word is the least possible, the efficiency has improved from 33% (2/6) to 61% (8/13). The example of unrolling the loop again is as follows:

```
for ( ; size>3 ; size-=4) {
    dst[0] = src[0];
    dst[1] = src[1];
    dst[2] = src[2];
    dst[3] = src[3];
    dst += 4;
    src += 4;
}
if (size<2) {
    if (size==1) {
        dst[0] = src[0];
    }
} else {
    if (size==2) {
        dst[0] = src[0];
        dst[1] = src[1];
    } else {
        dst[0] = src[0];
        dst[1] = src[1];
        dst[2] = src[2];
    }
}
```

Unfortunately, as the loop gets more and more unrolled, the cleanup code after the loop gets more and more complicated. For massively unrolled loops, the cleanup code might best be handled with a switch statement:

```
for ( ; size>15 ; size-=16 ) {
    dst[0] = src[0];
    dst[1] = src[1];
    ...
    dst[15] = src[15];
    dst += 16;
    src += 16;
}
switch(size) {
    case 15: dst[14] = src[14];
    case 14: dst[13] = src[13];
    ...
    case 1: dst[0] = src[0];
}
```

You will need to keep in mind just how the code will be used. If you plan to copy large amounts of data (like on the order of *Kwords*), then it's perfectly reasonable to unroll your loops 16 or 32 times. If you plan to only copy a handful of words, then extreme loop unrolling can actually hurt your performance. The loop should only be unrolled to the extent that a typical use will execute at least a few iterations.

### *Reduce the Cost of Multiple Clause If Tests*

In C/C++, `if` tests treat the logical operations `&&` and `||` specially when performing expression evaluation. The compiler produces code that will only continue to evaluate the expression as long as the result is not known. For example, the code below tests for `b==3` which will only occur if `a==1`. If `a!=1`, then the expression cannot possibly be true regardless of what `b` is.

```
if ( (a==1) && (b==3) ) {
    /* do something */
}
```

To the compiler, the code above looks like:

```
if (a==1) {
    if (b==3) {
        /* do something */
    }
}
```

Likewise, the following code:

```
if ( (a==1) || (b==3) ) {
    /* do something */
}
```

looks to the compiler as follows:

```
if (a==1) {
    /* do something */
} else if (b==3) {
    /* do something */
}
```

So when using `&&`, you should put the sub-expression most likely to fail at the beginning of the expression. When using `||`, put the sub-expression most likely to succeed at the beginning. This will reduce the average number of `if` tests your code executes.

Sometimes you will have a long list of `&&` separated `==` expressions which you think will frequently succeed. This commonly happens when you are checking current state versus cached state. By avoiding the use of `&&`, you can reduce the number of `if` tests by using integer math. For example, the following code:

```
if ( (v1->a == v2->a) && (v1->b == v2->b) && ... )
```

can be transformed to:

```
if ( !( (v1->a - v2->a) | (v1->b - v2->b) | ... ) )
```

It's worth noting that the recommended code will be slower than the original code if, for example, `v1->a != v2->a`. This technique should only be used when all clauses are expected to frequently be true.

Similar techniques can be used with `||` separated `if` tests. For example, the following code:

```
if ( (v1->a != v2->a) || (v1->b != v2->b) || ... )
```

can be transformed to:

```
if ( (v1->a - v2->a) | (v1->b - v2->b) | ... )
```

Using fast greater-than or less-than operators takes a bit more effort but is still useful. For example, the following code:

```
if ((x>=xmin) && (x<=xmax) && (y>=ymin) && (y<=ymax))
```

can be transformed to:

```
if (!((x-xmin) |
      (xmax-x) |
      (y-ymin) |
      (ymax-y)) >> 31))
or:
if (!((x-xmin) |
      (xmax-x) |
      (y-ymin) |
      (ymax-y)) & 0x80000000))
```

This takes advantage of the sign bit from the subtractions to do branch free comparisons. Again, these techniques should *only* be used when all clauses of an `if` test are likely to be evaluated.

## Avoid Using Malloc/Free and New/Delete

Allocating and freeing memory is an expensive operation. If a section of code on a performance critical path requires its own temporary space, try to either allocate it on the stack or cache it somewhere.

Allocating space on the stack is easiest when you know in advance how much space you will need and the amount is reasonably small (like space for a handful of 4x4 transforms). If you don't know how much space you will need at compile time, but you do know that it's small, you can try using `alloca()` instead of `malloc()`. The `alloca()` function gives you an amount of memory by bumping the stack pointer. This method of memory allocation should be used with caution since it will fail if you exceed the stack limit<sup>1</sup>. Since this method of allocation uses the stack, there is an implicit free when you leave the calling function.

1. Not only is `alloca()`'s behavior on failure undefined, the man page strongly discourages its use.

If you need to `malloc/new` space, try to cache a pointer to that space in whatever structure is both handy and likely to be around the next time through the code. You will need to check each time that you have enough space (and if you don't, free the old space and allocate a bigger chunk), but this is very cheap when compared to the costs of memory allocation.

Sometimes neither of these schemes is appropriate. If this is the case, try to minimize the number of allocations you do. Calculate the total size you will need, allocate one big chunk, and set your pointers to the appropriate offsets. It's worth noting that this performance recommendation is in direct opposition with object-oriented design principles. You will need to decide before you begin which is more important to you.

## *Cache Whatever You Need*

Another basic technique is caching values that you need. If you think it's likely for the same path to be taken through the code many times in a row, then look for calculated or constructed values that can be cached for future use. This applies particularly to requests that involve context switching (for example, system calls or Xlib inquiries). Although caching is a useful technique, you need to keep in mind the complexity of invalidating your caches. Don't leave this crucial aspect out of your design phase.

## *Preserve Batching*

When a library is handed data from the application, it is almost always a bad idea for the library to break it into smaller pieces. Not only does breaking up data make the library code more complicated and harder to understand, it is virtually guaranteed to reduce performance<sup>1</sup>. The only way to increase the batching factor beyond what the application gives you is to go through a copy operation. A low-level routine should not have to perform a copy just because a high-level routine broke up data.

---

1. An exception to this rule is if you have a multiprocessor system. In this case, it may be best to hand whatever data you've got to an idle processor. Even in cases such as this, before and after measurements should take place to ensure that the performance does actually go up.

## *Keep Parallelism As High As Possible*

In an immediate mode accelerated graphics environment, it is critical to consider the parallelism between the CPU and the accelerator to get anywhere near maximum performance. Some accelerator attributes will require the hardware pipeline to be empty while other attributes will not. You must look closely at the attributes that require the pipeline to be empty. As soon as one of these attributes come down, flush any outstanding data to the accelerator. Attempt to delay sending the attribute as long as possible. That may mean you should return to the application after setting some state to indicate that an attribute change is pending. On the next call to your library, wait for the accelerator to become idle, send the attribute, and finally process whatever the current call was.

## *Avoid Using Global Variables*

Because this is the age of dynamic libraries, all code must be relocatable. This also applies to global variables (local variables are kept on the stack and are therefore easy to locate). This need forces global variables to be referenced via a table indirection. Depending on whether your library is compiled `-pic` or `-PIC`, this will either be a single level or double level indirection. In a multi-threaded environment, global variables have the additional overhead of needing protection via locks. If you must use global variables, minimize their usage by creating local variables that point to the globals. The following code:

```
int    pt_size = global_data.pt_size;
...
float  *pt = global_data.pt;
...
float  *colors = global_data.vertex_colors;
```

would run faster as:

```
gd     *lgd = &global_data;
int    pt_size = lgd->pt_size;
...
float  *pt = lgd->pt;
...
float  *colors = lgd->vertex_colors;
```

In the second case, the indirection is only performed once per function instead of every time the global data is referenced.

## *Reduce Function Call Depth*

Depending on your hardware, function calls can be relatively cheap or expensive. Regardless, they are never free. An effort should be made to keep the function call depth to a reasonable limit. Not only can this result in less instructions executed, but it will reduce the number of code pages you touch so that your working set will be smaller. However, it is not recommended that you have complicated functions in order to reduce the number of code pages. As you are designing your code, just bear in mind the cost of function calls.

SPARC is optimized around register windows. Programs that are well-behaved in their function call depth will benefit from this and those that are not will suffer when their register windows frequently spill to memory.

x86 has relatively expensive function calls. Even ignoring the issues of pushing and popping parameters from the stack, the instructions `call`, `leave`, and `ret` take 3, 5, and 5 cycles, respectively on a 486 (the cycles are 7, 4 and 10 on a 386).<sup>1</sup>

## *Use Fixed Point Arithmetic*

Depending on the following listed criteria, it may be advantageous to convert your floating point data to fixed point. These criteria are as follows:

- Relative speeds of integer and floating point code on your hardware
- Whether your hardware uses super-scalar technology
- How much precision you need

This is particularly true if you are walking scanlines and you would like to avoid a floating point to integer cast for every pixel. Addition and subtraction are the two most common operations on fixed point numbers (since floating point multiplication/division is usually faster than integer multiplication/division).

---

1. Cycle counts from Microsoft's 80386/80486 Programming Guide, Ross P. Nelson, 1991.



## *Exploit the Math That Your Hardware Does Well*

If you know which specific platform your code will be running on, you can exploit the hardware to its fullest. If your code needs to run well on a variety of hardware, you may be forced to use the lowest common denominator. If this is the case, avoid using square root, integer multiplication/division/remainder, and to a lesser extent, floating point division. It is safe to assume that you will have fast addition and subtraction of both integer and floating point data. In addition, floating point multiplication is fast.

Some examples of things you can do are the following:

- Avoid using square root if you don't need to. For example, don't normalize vectors if you only need them for backface culling. Put an `if` test in your vector code to normalize only when necessary.
- If a variable will be used to divide two or more other variables, calculate its reciprocal once and use that to multiply with the other variables. Sometimes you can avoid both integer and floating point division of variables by multiplying other variables. For example:

```
if (a/10 >= b) and if (a/10.0 >= b)
```

can be replaced with:

```
if (a >= b*10) and if (a >= b*10.0)
```

Notice that the integer technique works with “`>=`” but not with “`>`”.

- If you know something about one of the operands of an integer multiplication, you may be able to use shifts and adds to get the result. For example, if you know that one operand will always be between 0 and 15, then use a switch with 16 cases that multiplies the other operand by a constant. Be sure to check to make sure the compiler turns this into a series of shifts and adds.
- The only kind of fast integer division and remainder is when the divisor is a power of 2. If you have such a divisor, then code using either shifts and ands, or verify that the compiler is smart enough to notice.

## *Use Single Precision Floating Point Constants*

ANSI C/C++ dictates that floating point constants by default are double precision. This affects your code in several ways:

- Two words of data are loaded from memory instead of one for every floating point constant.
- Variables and temporary values may undergo a conversion to double precision (for example, more instructions).
- Fewer floating point registers are available because you have double precision copies of single precision data.
- Expression evaluation is done using double precision instructions, which are potentially slower than single precision instructions.

Fortunately ANSI allows you to keep all your floating point constants in single precision by adding an `f` suffix. For example, change:

```
if (val < 0.0)
```

to:

```
if (val < 0.0f)
```

to get 0.0 to be single precision. Note that if you are using `cc`, you will also need to apply the `-fsingle` compile line option to get single precision expression evaluation. You can think of the `f` suffix as merely registering with the compiler that the constant's data type is `float` and not `double`.

## *Avoid Careless Use of the Stack*

Use the stack sparingly. RISC CPUs tend to have an abundance of general purpose registers that are quite effective in increasing performance. Keeping your function's local variables in these registers can dramatically increase the speed of your code. On SPARC, look for references to the frame pointer (`%fp`) in your performance critical functions. Pay close attention to the references inside your inner loops. For the most performance-critical functions, it is an achievable goal to have absolutely no references to `%fp`.

Removing references to `%fp` is not easy. You may find that you need to break up a function into many smaller, specialized functions. Frequently, however, you will be able to tune the code so that it can be easily processed by the compiler. Creating new local variables can be used to move `%fp` references outside of a loop.

Be advised that declaring variables to be of type `register` does not guarantee that they will actually be in a register. The `register` keyword is only a hint to the compiler. Different compilers (and even different versions of the same compiler) will consider this keyword differently. It is perfectly legal for a compiler to completely ignore this hint.

Experiment with changing the code to see just what it is that your compiler needs. This level of tuning requires looking at the assembly output of your compiler. Every compiler has its quirks, and your task is to figure out these quirks.

## *Optimized Leaf Functions*

CPUs with register windows typically have a much lower function call cost than CPUs that don't have register windows. Above and beyond this, register windows support an even faster kind of mini-function called an *optimized leaf function*. The idea is that if a function only uses windowed *out* registers (no local or floating point registers, and no stack or frame pointers), and calls no other functions, then the function can operate using the caller's register window and stack frame. The benefits of optimized leaf procedures is a savings of one or two instructions per call plus the possibility of not overflowing the register windows.

## *Try to Minimize Loads*

On high-clock rate RISC machines, loads are much more expensive than stores. This is because loads require a round trip message. First, the request is made by the CPU for some data, and at some later time the data is given to the CPU. Stores are faster because the CPU simply issues a request for the data to be stored, and the memory subsystem worries about the rest. Loads can also be slower because they may have to wait for the store buffer to drain (see "Cluster Loads and Cluster Stores" on page 434). CPU caches exist to alleviate the problems associated with loads, but the cache will never get a 100% hit rate,

nor will it help with uncachable data like a device's registers. The penalty for a cache miss is high enough to factor it into a design. As CPU speed continues to go up, this penalty will get higher.

Techniques for minimizing loads have already been brought up, but it is worth repeating here. Make sure that loops have local variables that directly reference the data you are interested in. Don't have code like:

```
for ( i=0 ; i<size ; i++ ) {
    ...
    sum += a->b[i];
    ...
}
```

This should be changed as follows:

```
bptr = a->b;
for ( i=0 ; i<size ; i++ ) {
    ...
    sum += bptr[i];
    ...
}
```

Also keep pointers to global variables around if possible. Look for stack references and minimize their number. Loading data from across a bus is particularly expensive, so you should try to limit this process as much as possible.

## Cluster Loads and Cluster Stores

Current RISC hardware handles back-to-back loads and back-to-back stores well, but does not handle load-store-load-stores well. This is partly due to the cache. Every time you store a new value into the cache, you run the risk of invalidating some data that you're about to load. As caches get more associative, this risk goes down, but it never goes away.

Most processors also have what is known as a store buffer. This is typically a small FIFO queue that fills up with data to store if the memory subsystem is busy. A CPU may need to wait for the store buffer to empty before a load can be issued. The problem here is basically parallelism. The ideal is to have your CPU and memory subsystem doing productive work at all times.

If you were to rewrite the body of the 4-way unrolled copy loop below:

```
for ( ; size>3 ; size-=4) {  
    dst[0] = src[0];  
    dst[1] = src[1];  
    dst[2] = src[2];  
    dst[3] = src[3];  
    dst += 4;  
    src += 4;  
}
```

to cluster loads and cluster stores, it would look like:

```
for ( ; size>3 ; size-=4) {  
    t0 = src[0];  
    t1 = src[1];  
    t2 = src[2];  
    t3 = src[3];  
    dst[0] = t0;  
    dst[1] = t1;  
    dst[2] = t2;  
    dst[3] = t3;  
    dst += 4;  
    src += 4;  
}
```

This compiles to the same number of assembly instructions, but the loads and stores are handled in blocks rather than interleaved for every word copied. As tends to happen, this code uses more local variables (and also more registers) to force the compiler to do what you want in the order you want. Even though you are programming in C, the source code compiles almost line for line to assembly code, and the local variables tend to map one to one with hardware registers.

### *Use Double Word Loads and Stores*

SPARC supports the concept of 64-bit quantities through C and C++. This can be advantageous for setting and moving blocks of data. Each double word load/store instruction takes the place of two single word load/stores. Depending on the characteristics of the memory subsystem, you may be able to achieve an almost perfect 2:1 speedup (going over an I/O bus is such a case).

Double word load/stores can only be used on double word-aligned data. For setting or clearing a block of data, this is easily handled by testing the starting address and possibly writing out a single word of data before entering the main double word loop. For copy operations, there is the additional complexity of the source address being double-aligned, but the destination address is not (and vice versa). In general, there is no way in C to exploit double word load/stores in this situation. You would need to have an assembly language routine to do it. However, if you are writing data to a device input buffer, you may be able to write a one word `NO_OP` directive to get the source and destination address alignments synchronized.

To convert the body of our unrolled loop to use double word load/stores requires casting things properly, as shown below:

```
for ( ; size>3 ; size-=4) {
    d0 = *(double*)(src+0);
    d1 = *(double*)(src+2);
    *(double*)(dst+0) = d0;
    *(double*)(dst+2) = d1;
    dst += 4;
    src += 4;
}
```

Of course, you should have your `src` and `dst` declared to be double (or long) to improve the code readability. Also, the *ProWorks* compiler needs to have the `-dalign` flag to use double word load/stores.

## Be Cache-Aware

Certain types of algorithms need to take into account the hardware caches present in the systems they will run on. If you are writing code that accesses large amounts of data (memory rasters, Z-buffers, texture maps), you should bear in mind how the hardware cache will affect the performance of your code. Try to keep your data references bounded within small local regions. Also, when allocating space for data structures, try to keep adjacent data from mapping to the same cache line. For example, if you need 1024-byte scanlines, allocate 1152 bytes per scanline so that pixel X,Y doesn't map to the same cache line as pixel X,Y+1.

## Compiler Options

A general recommendation is to know the optimizations that your compiler supports. Although compiler options will vary depending on your code and the system, some possible options are listed below. Check your reference pages for more information.

Table A-2 Compiler Options

cc	CC
-xcgXX	-cgXX
-dalign	-dalign
-fast	-fast
-fsingle	
	-ispace vs -ispeed
-xlibmil	-libmil
-native	-native
-xO	-O
-K pic vs -K PIC	-pic vs -PIC
	-Qoption fbe -cgXX
-xunroll	

≡ A

---



## *Changes to the XGL Graphics Porting Interface*

---

***B*** 

This appendix provides information on the differences between the previous XGL graphics porting interface (GPI) and the current XGL GPI. It lists and briefly describes additions, changes, and deletions to the GPI. For more information in the device-independent changes, see the *XGL Architecture Guide*. For current information on XGL operators, attributes, and data structures, see the *XGL Reference Manual*.

### *Changes in Rendering Architecture*

This section lists changes to the rendering architecture.

#### *Interface Manager Removed*

To simplify the interactions between the device pipeline and the device-independent code, the interface manager object has been removed from the XGL architecture. Pipeline renderers are now called via function pointers in the `opsVec[ ]` array, which is defined in the `XglDpCtx` object. The pipeline must set up a pipeline-specific version of the `opsVec[ ]` array to point to the rendering functions for the primitives that it accelerates. If the device pipeline does not implement a function, the `opsVec[ ]` value for that function will call the software pipeline by default. See page 44 in Chapter 3, “Pipeline Framework” for information on setting up the `opsVec[ ]` array. Be sure to remove the interface manager header files from your pipelines.

Because the interface manager has been removed, the device pipelines no longer call the the interface manager to access other primitives from within a given primitive. Therefore, note the following about calling the software pipeline or another LI primitive.

- Device pipelines call the software pipeline directly. Previously, the device pipeline called the software pipeline through the interface manager in these cases:
  - If the device pipeline had not implemented a renderer, it returned a value of 0 to a rendering call.
  - The device pipeline could call the interface manager with a return value of 0 in certain cases within an implemented primitive.
  - The device pipeline could call the software pipeline for partial processing of data by calling the interface manager LI function with the software override flag set to `TRUE`.

A pipeline return value of 0 or a pipeline call to the interface manager with the software override flag set to `TRUE` caused the interface manager to call the software pipeline.

At this release, the device pipeline is responsible for calling the software pipeline directly, as:

```
swp->lilMultiPolyline(api_bbox, api_num_plists, api_pt_list);
```

- Device pipelines call other LI functions through the `opsVec[]` array, or they can call their own renderers directly. Previously, if a pipeline created a Gcache from within `lilMultiSimplePolygon()` to handle certain polygon cases, the pipeline called `itfMgr->lilDisplayGcache(gcache)` to render the polygons. Now, any call of `itfMgr->{primitiveCall}` should be changed to a call through the `opsVec[]` array or a direct call.

In general, you would want to use the `opsVec[]` array to call other primitives, since this architecture has been set up to be advantageous to subsequent primitive calls. In some device-dependent cases, calling a primitive directly might be faster. For example, for rendering polygons with edges, calling `lilMultiPolyline()` directly to draw the edges may be faster for some devices than using the `opsVec[]` array.

Note that when calling another `XglDpCtx` primitive through the `opsVec[]` array, the call should include an extra parameter, `gen_punt = FALSE`, in order for backing store to work correctly.

For instructions on setting opsVec pointers in the XglDpCtx object, see Chapter 3, “Pipeline Framework”.

### *Primitive Return Types Changed*

At this release, primitives are no longer virtual functions, and LI-1 and LI-2 primitive calls no longer return values. Thus, for example,

```
virtual int XglDpCtx3d::lilMultiArc(XglConicData* arc_data)
```

has become:

```
void XglDpCtx3d::lilMultiArc(Xgl_arc_list* arc_list)
```

Note that LI-3 functions have not changed.

### *Primitive Arguments Changed at LI-1*

The arguments to LI-1 primitives have been changed to pass the API data directly to the device pipeline rather than through the XglPrimData object. Each primitive function is called directly with the application data. For example, the application calls `xgl_multipolyline()` as:

```
xgl_multipolyline(Xgl_ctx ctx, Xgl_bbox* bbox,  
                  Xgl_usgn32 numPtLists, Xgl_pt_list pl[])
```

The corresponding call to the device pipeline was previously:

```
lilMultiPolyline(XglPrimData *pd);
```

It now is this:

```
lilMultiPolyline(Xgl_bbox* bbox, Xgl_usgn32 numPtLists,  
                  Xgl_pt_list* pl)
```

See Chapter 3, “Pipeline Framework” for the current LI-1 primitive arguments.

### *Constructor Change*

The XglDpCtx constructor calling parameters have changed. Previously, its calling parameters were:

```
XglDpCtx{2,3}d(context)
```

At this release, the calling parameters are:

```
XglDpCtx{2,3}d(dp_dev->getDevice(), context)
```

## *Changes in State Handling*

As with the interface manager object, the update tables have been removed to optimize the internal architecture. In place of the update tables, the pipeline should create the following two functions and insert pointers to them in the `opsVec[]` array:

- `objectSet()` – Function that passes information on Context attribute changes to the device pipeline when changes occur.
- `messageReceive()` – Function that passes the device pipeline information on attributes changes in objects other than the Context.

You can copy these functions from the GX sample pipeline and update the XGL Context types in the `switch` statement with the Context types appropriate for your hardware.

As an alternative, the pipeline can retrieve Context attributes every time it renders. However, for optimized performance, the `objectSet()` architecture is recommended for LI-1 primitives.

Be sure to remove the update table header files from your existing renderers. In addition, remove all references to update table masks. For information on Context state handling at this release, see Chapter 5, “Handling Changes to Object State”.

## *Derived Data Change*

Derived data has the same interface at this release except for the `updateTableChanged()` function. Previously, a device pipeline called the `udTable.updateTableChanged()` function to determine whether changes to derived data occurred. Because the update tables have been removed, the view group function `changedComposite()` has been modified to incorporate the quick test for derived data changes that the update table provided. Therefore, `viewGrpItf->changedComposite()` is now the first indication that derived data may have changed. For information on the derived data mechanism, see Chapter 7, “View Model Derived Data”.

## *Application Data Passed Directly to Pipelines*

As mentioned above, the `XglPrimData` object is no longer used to process data from the application at LI-1. LI-1 primitive functions now receive actual API data instead of the preformatted data in the `XglPrimData` objects. Because of this, arguments for LI-1 primitive functions have changed. Be sure to remove all references to `XglPrimData` from LI-1 primitives.

### *Utility Arguments Changed*

The calling arguments for the utilities that took `XglPrimData` objects as an argument have changed. Table B-1 lists the changed utilities. These utilities now take the API data in place of `XglPrimData` object data.

*Table B-1* Changed Utilities for XGL 3.1

Changed Utilities
<code>XgliUtComputeFn</code>
<code>XgliUtComputeFnReverse</code>
<code>XgliUtComputeIndepTriFn</code>
<code>XgliUtComputeIndepTriFnPl</code>
<code>XgliUtComputeMspFn</code>
<code>XgliUtComputePolygonFn</code>
<code>XgliUtComputeQuadMeshFn</code>
<code>XgliUtComputeTstripFn</code>
<code>XgliUtComputeTstripFnPl</code>
<code>XgliUtComputeTstarFn</code>
<code>XgliUtComputeTstarFnPl</code>
<code>XgliUtComputeVnReverse</code>
<code>XgliUtMellaToPline</code>
<code>XgliUtModelClipMarker</code>
<code>XgliUtModelClipMpline</code>
<code>XgliUtModelClipMspg</code>
<code>XgliUtPdModelClipPgon</code>

*Table B-1* Changed Utilities for XGL 3.1

---

**Changed Utilities**

---

XgliUtModelClipTstrip

XgliUtVertexOrientation

XgliUtClassifyMsp

XgliUtClassifyPgon

---

If there is interest in accelerating part of the NURBS curve or surface, or in the algorithms, refer to the following papers. Be aware that the coordinate system changes in different situations.

- Abi-Ezzi, Salim. "The Graphical Processing of B-splines in a Highly Dynamic Environment," Rensselaer Polytechnic Institute, RDRC-TR 89001, Troy, New York, May 1989.
- Abi-Ezzi, Salim and Leon Shirman. "The Tessellation of Curved Surfaces Under Highly Varying Transformations," in Proc. Eurographics 1991, F. H. Post and W. Barth, eds., Eurographics Association, Elsevier Science Publishers B.V. North Holland, 1991.
- Abi-Ezzi, Salim and Leon Shirman. "The Scaling Behavior of a Viewing Transformation," accepted for publication in *IEEE Computer Graphics and Applications*, 1992.
- Abi-Ezzi, Salim and Michael Wozny. "Factoring a Homogeneous Transformation for a More Efficient Graphics Pipeline," in *Computer Graphics Forum*, Vol. 9, 1990.
- Abi-Ezzi, Salim, and Srikanth Subramaniam. "Compilation for Fast Dynamic Tessellation of Trimmed NURBS Surfaces," Unpublished, 1993.
- Farin, Gerald. *Curves and Surfaces for Computer Aided Geometric Design*, Second Edition, Academic Press, Inc., San Diego, CA, 1990.
- Garey, M., D. Johnson, F. Preparata, and R. Tarjan. "Triangulating a Simple Polygon," in *Information Processing Letters*, Vol. 7, No. 4, June 1978.

- 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.
- *Solaris XGL 3.0.1 Programmer’s Guide*, part number 801-4120-10, Sun Microsystems, Inc.
- Rockwood, Alyn, Kurt Heaton, and Tom Davis. “Real-Time Rendering of Trimmed Surfaces,” in *Computer Graphics*. Proceedings of Siggraph 1989, Vol. 23, No. 3, July 1989.
- Shirman, Leon and Salim Abi-Ezzi. “The Cone of Normals for Fast Processing of Curved Patches,” Submitted for publication, 1992.



# Index

---

## A

- accumulation buffer, 53, 56
- addPickToBuffer(), 127
- assignCurStrokeAsEdge(), 128
- assignCurStrokeAsLine(), 128
- assignCurStrokeAsMarker(), 128
- assignCurStrokeAsSurfBack(), 129
- assignCurStrokeAsSurfFront(), 128
- assignCurStrokeAsText(), 128

## B

- backing store
  - and double buffering, 67
  - device pipeline support, 65
  - overview, 10

## C

- changedComposite(), 164
- checkLastPick(), 128
- clearComposite(), 165
- clip lists, 186, 191
- clipChanged(), 198
- Color Map object
  - object interfaces, 146
- Context object

- getting attribute values, 123
- object interfaces, 127
- context switching, 9, 115
- coordinate systems, 154, 168, 180
  - See also* derived data
- copyBuffer(), 40
- copyConvert(), 143
- createDpCtx(), 39
- createDpDev(), 37
- current coordinate system, 180
- current stroke pointer, 109

## D

- data storage
  - accessing data at LI-1, 75
  - accessing data at LI-2, 82
  - conic data, 85 to 89
  - facet data, 76
  - in the software pipeline, 79
  - level data, 80
  - overview, 74
  - pixel data, 89 to 94
  - point data, 82 to 84
  - rectangle data, 85 to 89
- dbDisplayComplete(), 198
- dbDisplayWait(), 198
- dbGetWid(), 198

---

- dbUnGrab(), 203
- DC offset values, 112
- DDK (Device Driver's Kit), 8
- deallocate(), 93
- Denizen test suite, 20
- depth cue reference planes, 175
- derived data
  - changes of derived items, 164
  - coordinate systems, 154
  - design goals, 151
  - eye vector, 156
  - getting boundaries, 170
  - getting eye vector, 173
  - getting lights, 172
  - getting model clip planes, 174
  - getting transforms, 169
  - lights, 156, 172
  - transforms, 154
  - view cache object, 157
  - view clip bounds, 156
  - view concern object, 158
  - view group configuration object, 157
  - view group interface object, 158
  - view model, 150
- Device Driver's Kit (DDK), 8
- Device object, 29
  - initialization, 42
  - object interfaces, 131
- device orientation, 52
- device pipeline
  - accessing point data at LI-1, 75
  - accessing point data at LI-2, 82
  - adding member data to a class, 61
  - error reporting, 23 to 27
  - getting attribute values, 120
  - getting model clip planes, 174
  - getting transforms, 169
  - getting view clip bounds, 170
  - locking the window for
    - rendering, 186, 192
  - managing clip list changes, 186
  - managing window system
    - resources, 195
  - multiple frame buffers, 34
  - multiple windows, 38
  - naming conventions, 51
  - pipeline context class (XglDpCtx), 43
    - to 66
  - pipeline device class (XglDpDev), 38
    - to 42, 52 to 57
  - pipeline initialization, 42, 190
  - pipeline library class (XglDpLib), 33
    - to 35
  - pipeline loading, 58
  - pipeline manager class
    - (XglDpMgr), 36 to 37
  - required classes, 30
  - return values, 66
  - summary of functions, 68
  - xgl\_create\_PipeLib(), 32
- DGA, 9, 184
  - winBboxinfop(), 205
  - winDbInfop(), 205
- dithering
  - lookUpDitherValue(), 146
  - lookUpInternalDitherAddress(), 146
  - lookUpInternalDitherValue(), 146
- dlsym(), 32
- DMA devices, 78
- double buffering, in hardware, 195
- dynamic linking, 2
  - dlsym(), 32

## E

- error reporting, 23 to 27
- external files, 21
- eye vector, 156
- eye vectors, 173

## F

- fast clear sets, 200
- fillRectangle(), 92
- fonts
  - stroke font object interfaces, 136
- frame buffers, multiple, 34, 36

---

## G

gamma values, 131  
getAccumBufferDepth(), 54  
getAccumBufferPixRect(), 136  
getActualData(), 132  
getActualDataSize(), 132  
getActualDescription(), 133  
getActualOffset(), 133  
getApiData(), 75  
getBackTexturing(), 130  
getBbox(), 88  
getCenterPtr(), 88  
getClass(), 203  
getClipStat(), 198  
getCmap(), 145  
getCmapDrawable(), 146  
getColorTable(), 146  
getConicDataType(), 88  
getConicType(), 88  
getCosAngle2(), 132  
getCurCoordSys(), 180  
getCurrentLevel(), 84, 88  
getCurrentLevelData(), 84, 88  
getCurrentStroke(), 128, 129  
getDcOrientation(), 52, 131  
getDepth()  
    in XglDpDevWinRas, 54  
    in XglPixRect, 92  
getDescriptor(), 203  
getDescriptors(), 130, 134  
getDevFd(), 198  
getDevice(), 198  
getDeviceName(), 198  
getDoPixelMapping(), 134  
getDpDev(), 131  
getDpMgr(), 34  
getDrawable(), 131  
getElement(), 133  
getExpectedFlagValue(), 112  
getFaceAttrs(), 84  
getFacetList(), 84  
getFlag(), 137, 142  
getFlagMask(), 112  
getFlagPtr(), 88  
getFrontTexturing(), 129  
getGammaInversePowerTable(), 131  
getGammaPowerTable(), 131  
getGammaValue(), 52, 131  
getHeight(), 92  
getImageBufferPixRect(), 135  
getImgBufLineBytes(), 136  
getInverseMapperHasBeenSet(), 147  
getIsFontLoaded(), 136  
getIsotropicScale(), 143  
getLength(), 133  
getLevelData(), 84, 88  
getLineBytes(), 93  
getLockType(), 203  
getMajorAxisPtr(), 88  
getMapperHasBeenSet(), 147  
getMatrix(), 143  
getMatrixDouble(), 143  
getMatrixFloat(), 142  
getMatrixInt(), 143  
getMaxZ(), 52  
getMemberRecord(), 138, 142  
getMemoryAddress(), 93  
getMergeClipList(), 199  
getMergeClipListCount(), 199  
getMergeClipMask(), 199  
getMinorAxisPtr(), 88  
getNegDirection(), 132  
getNewFramePlaneMask(), 127  
getNorm(), 143  
getNormInverse(), 143  
getNumConics(), 88  
getNumPointLists(), 84  
getNumRects(), 89  
getParallelProj(), 171  
getPipeName(), 203

---

getPlaneMaskMask(), 146  
getPointLists(), 84  
getProcessFlags(), 84  
getRadiusPtr(), 88  
getRealColorType(), 54, 199  
getRealPlaneMask(), 127  
getRealRenderBuffer(), 127  
getRenderFlags(), 84  
getRotAnglePtr(), 88  
getSfontData(), 136  
getSfontInst(), 136  
getStartAnglePtr(), 88  
getStartPointPtr(), 89  
getStartSeg(), 133  
getStartSegRemain(), 133  
getStopAnglePtr(), 89  
getStopPointPtr(), 89  
getSurfAttr(), 127  
getSurfBackAttr3d(), 129  
getSurfBackFaceAttr(), 129  
getSurfBackFaceAttr3d(), 129  
getSurfFrontAttr3d(), 129  
getSurfFrontFaceAttr(), 127  
getSurfFrontFaceAttr3d(), 129  
getSwAccumBuffer(), 56, 135  
getSwp(), 128  
getSwZBuffer(), 56, 135  
getType(), 203  
getUserClipList(), 200  
getUserClipListCount(), 200  
getValue(), 92  
getValueByPointer(), 92  
getViewCanonical(), 171  
getViewGrp(), 128  
getWid(), 199  
getWidth(), 92  
getWindowDepth(), 199  
getWindowHeight(), 199  
getWindowWidth(), 199  
getWindowX(), 199

getWindowY(), 199  
getWrapOriginX(), 92  
getWrapOriginY(), 92  
getWrappedValue(), 92  
getWsClipList(), 200  
getWsClipListCount(), 199  
getZBufferPixRect(), 135  
global state object, 34, 58  
grabDrawable(), 204  
grabFCS(), 200  
grabRetainedWindow(), 203  
grabStereo(), 201  
grabWids(), 200  
grabZbuf(), 200

## I

inquire(), 37  
isMemory(), 92

## L

lights, 156, 172  
line patterns, 132  
line-specific attributes, 106  
linking, dynamic, 2  
lock functions, 191  
lookUpDitherValue(), 146  
lookUpInternalDitherAddress(), 146  
lookUpInternalDitherValue(), 146

## M

markers, 133  
matchDesc(), 204  
matrices  
    getMatrix(), 143  
    getMatrixDouble(), 143  
    getMatrixFloat(), 142  
    getMatrixInt(), 143  
messageReceive(), 105  
model clip planes, 174  
multipolyines

---

- expected flag value, 111
- flag mask, 111
- primitives rendering as, 106
- stroke types, 106

## N

- naming conventions, 51
- naming conventions for internal attributes, 121
- needRtnDevice(), 56
- normals, 143

## O

- objectSet(), 70, 110, 122
- ops\_vector function array, 21, 44, 45
- opsVecDiDefault function array, 48

## P

- performance tuning
  - different performance tools, 412
  - find the performance critical paths, 410
  - selecting good benchmarks, 413
  - suggested techniques for faster code, 416
  - tuning the performance critical paths, 415
- picking
  - addPickToBuffer(), 127
  - checkLastPick(), 128
- pipeline, *See* device pipeline or software pipeline
- pixel data, 89
  - XglPixRect, 90
- popCurCoordSys(), 180
- porting
  - choosing an interface layer, 14
  - implementing an LI-1 primitive, 16
  - testing the implementation, 20
- possible(), 204
- pushCurCoordSys(), 180

## R

- reallocate(), 93
- reassign(), 94
- receive(), 128
- resize()
  - in XglDpDevWinRas, 54
  - in XglDrawable, 204

## S

- setBackingStore(), 54
- setBufDisplay(), 55
- setBufDraw(), 55
- setBuffersRequested(), 54
- setBufMinDelay(), 55
- setCmap()
  - in XglDpDevMemRas, 57
  - in XglDpDevWinRas, 55
- setComposite(), 165
- setCurCoordSys(), 180
- setCursorRopFunc(), 201
- setDisplayBuffer(), 201
- setDoPixelMapping(), 134
- setImageBufferAddr(), 57
- setLineBytes(), 57
- setNumConics(), 88
- setNumRects(), 89
- setPixelMapping(), 55
- setReadBuffer(), 202
- setRectList(), 53, 204
- setRectNum(), 53, 204
- setSourceBuffer(), 53
- setStereoMode(), 56
- setSwAccumBuffer(), 53
- setSwZBuffer(), 53
- setValue(), 92
- setValueByPointer(), 92
- setWrapOriginX(), 92
- setWrapOriginY(), 92
- setWriteBuffer(), 202
- setZBufferAddr(), 57

---

- shared memory, 191, 205
- software cursors, 196, 201
- software pipeline
  - derived data, 150
  - level data, 79
- state changes
  - overview, 96
  - stroke groups, 106
- stereo imaging, 195
- stroke attributes, 110
- stroke group, 106
  - DC offset, 112
  - example, 109
  - expected flag value, 111
  - flag mask, 111
- stroke group attributes, 125
- syncRtnDevice(), 54, 136

## T

- texture mapping
  - lighting coefficients, 83
  - object interfaces, 130, 134
- Transform object
  - flag data, 137
  - getting object handle, 125
  - matrices, 143
  - member record, 137
  - object interfaces, 142
- transforms, 154, 169
  - See also* derived data
- transNormal(), 144
- transPt(), 143
- transPtList(), 144
- transUnitNormal(), 144
- transUnitNormalDouble(), 144

## U

- unGrabDrawable(), 204
- unGrabRetainedWindow(), 204
- update table
  - design issues, 114
  - stroke groups, 106

## V

- view clip bounds, 156, 170
- view concern objects, 159
- view model, 150
  - See also* derived data
- VIS\_GETIDENTIFIER, 58

## W

- WIN\_LOCK(), 187, 192, 197
- WIN\_UNLOCK(), 192, 198
- winBboxInfo(), 205
- winDbInfo(), 205
- window locking, 187
  - asynchronous devices, 188
  - immediate-rendering hardware, 188
  - performance implications, 192
- window system
  - See also* XglDrawable
  - clip list, 191
  - clip list updates, 191
  - creation of the XglDrawable, 186
  - fast clear sets, 205
  - locking the window, 186, 187
  - window ID, 198, 199
- windowIsClipped(), 202
- windowIsObscured(), 202
- winLock(), 187, 197
- winUnlock(), 197

## X

- XGL architecture
  - and the device pipelines, 11
  - overview, 11
- xgl\_create\_PipeLib(), 32
- XglCmap
  - getColorTable(), 146
  - getPlaneMaskMask(), 146
  - lookUpDitherValue(), 146
  - lookUpInternalDitherAddress(), 146
- XglConicData, 85 to 89
  - getCurrentLevel(), 88

---

- getCurrentLevelData(), 88
  - getLevelData(), 88
- XglContext
  - addPickToBuffer(), 127
  - checkLastPick(), 128
  - getNewFramePlaneMask(), 127
  - getRealPlaneMask(), 127
  - getRealRenderBuffer(), 127
  - getSurfAttr(), 127
  - getSurfFrontFaceAttr(), 127
- XglContext2d
  - assignCurStrokeAs...(), 128
  - getCurrentStroke(), 128
  - getViewGrp(), 128
- XglContext3d
  - assignCurStrokeAs...(), 129
  - getBackTexturing(), 130
  - getCurrentStroke(), 129
  - getFrontTexturing(), 129
  - getSurfBackFaceAttr(), 129
  - getSurfBackFaceAttr3d(), 129
  - getSurfFrontFaceAttr3d(), 129
- XglDevice
  - getCmap(), 145
  - getDpDev(), 131
  - getDrawable(), 131
  - getGammaInversePowerTable(), 131
  - getGammaPowerTable(), 131
  - getGammaValue(), 131
- XglDmapTexture
  - getDescriptors(), 130, 134
- XglDpCtx, 43 to 66
  - getting Context attribute values, 123
  - summary of interfaces, 68
- XglDpDev, 38 to 42
  - accessing the Device object, 145
  - and the XglDrawable, 186
  - copyBuffer(), 39
  - createDpCtx(), 39
  - getDcOrientation(), 52
  - getGammaValue(), 52
  - getMaxZ(), 52
  - summary of interfaces, 68
- XglDpDevMemRas
  - getAccumBufferDepth(), 57
  - getAccumBufferPixRect(), 57
  - getImageBufferPixRect(), 56
  - getZBufferPixRect(), 57
  - setCmap(), 57
  - setImageBufferAddr(), 57
  - setLineBytes(), 57
  - setZBufferAddr(), 57
- XglDpDevRaster
  - setRectList(), 53
  - setRectNum(), 53
  - setSourceBuffer(), 53
  - setSwAccumBuffer(), 53
  - setSwZBuffer(), 53
  - syncRtnDevice(), 54
- XglDpDevWinRas
  - getAccumBufferDepth(), 54
  - getDepth(), 54
  - getRealColorType(), 54
  - getSwAccumBuffer(), 56
  - getSwZBuffer(), 56
  - need RtnDevice(), 56
  - resize(), 54
  - setBackingStore(), 54
  - setBufDisplay(), 55
  - setBufDraw(), 55
  - setBuffersRequested(), 54
  - setBufMinDelay(), 55
  - setCmap(), 55
  - setPixelMapping(), 55
  - setStereoMode(), 56
- XglDpLib, 33 to 35
  - getDpMgr(), 34
- XglDpMgr, 36 to 37
  - and the XglDrawable, 186
  - createDpDev(), 37
  - inquire(), 37
- XglDrawable
  - clipChanged(), 198
  - creation, 186
  - dbDisplayComplete(), 198
  - dbGetWid(), 198
  - dbUnGrab(), 203
  - dpDisplayWait(), 198
  - getClass(), 203

---

getClipStat(), 198  
 getDescriptor(), 203  
 getDevFd(), 198  
 getDevice(), 198  
 getDeviceName(), 198  
 getLockType(), 203  
 getMergeClipList(), 199  
 getMergeClipListCount(), 199  
 getMergeClipMask(), 199  
 getPipeName(), 203  
 getRealColorType(), 199  
 getType(), 203  
 getUserClipList(), 200  
 getUserClipListCount(), 200  
 getWid(), 199  
 getWindowDepth(), 199  
 getWindowHeight(), 199  
 getWindowWidth(), 199  
 getWindowX(), 199  
 getWindowY(), 199  
 getWsClipList(), 200  
 getWsClipListCount(), 199  
 grabDrawable(), 204  
 grabFCS(), 200  
 grabRetainedWindow(), 203  
 grabStereo(), 201  
 grabWids, 200  
 grabZbuf(), 200  
 matchDesc(), 204  
 possible(), 204  
 rendering, 186  
 resize(), 204  
 services provided, 185  
 setCursorRopFunc(), 201  
 setDisplayBuffer(), 201  
 setReadBuffer(), 202  
 setRectList(), 204  
 setRectNum(), 204  
 setWriteBuffer(), 202  
 software cursors, 196  
 subclasses, 184  
 synchronizing window access, 191  
 unGrabDrawable(), 204  
 unGrabRetainedWindow(), 204  
 WIN\_LOCK(), 197  
 WIN\_UNLOCK(), 198  
 windowIsClipped(), 202  
 windowIsObscured(), 202  
 winLock(), 197  
 winUnLock(), 197  
 XGLHOME environment variable, 21  
 XGLI\_DC\_OFFSET\_BACK, 113  
 XGLI\_DC\_OFFSET\_FRONT, 113  
 XGLI\_DC\_OFFSET\_NONE, 112  
 XGLI\_PIPELINE\_CHECK\_VERSION(), 65  
 XGLI\_TRANS\_INVERSE\_VALID, 137  
 XGLI\_TRANS\_SINGULAR, 137  
 XglLevel, 79 to 83  
     getFaceAttrs(), 84  
     getFacetList(), 84  
     getNumPointLists(), 84  
     getPointLists(), 84  
     getRenderFlags(), 84  
 XglLight  
     getCosAngle2, 132  
     getNegDirection(), 132  
 XglLinePattern  
     getActualData(), 132  
     getActualDataSize(), 132  
     getActualOffset(), 133  
     getLength(), 133  
     getStartSeg(), 133  
     getStartSegRemain(), 133  
 XglListOfDpMgr, 190  
 XglMarker  
     getActualDescription(), 133  
 XglMipMapTexture  
     getElement(), 133  
 XglPixRect, 89 to 94  
 XglPrimData, 82 to 84  
     getApiData(), 75  
     getCurrentLevel(), 84  
     getCurrentLevelData(), 84  
     getLevelData(), 84  
     getProcessFlags(), 84  
 XglRaster  
     getDoPixelMapping(), 134  
     setDoPixelMapping(), 134



---

XglRasterWin  
    getSwAccumBuffer(), 135  
    getSwZBuffer(), 135  
XglRectData, 85 to 89  
XglSfont  
    getIsFontLoaded(), 136  
    getSfontData(), 136  
    getSfontInst(), 136  
XglTransform  
    getFlag(), 142  
    getIsoTropicScale(), 143  
    getMatrix(), 143  
    getMatrixDouble(), 143  
    getMatrixFloat(), 142  
    getMatrixInt(), 143  
    getMemberRecord(), 142  
    getNorm(), 143  
    getNormInverse(), 143  
    transNormal(), 144  
    transPt(), 143  
    transPtList(), 144  
    transUnitNormal(), 144  
    transUnitNormalDouble(), 144  
XglViewCache2d, 157  
XglViewCache3d, 157  
XglViewConcern2d, 157  
XglViewConcern3d, 157  
XglViewGrp2dConfig, 157  
XglViewGrp2dItf, 157  
XglViewGrp3dConfig, 157  
XglViewGrp3dItf, 157

## **Z**

Z-buffers, hardware, 195

