HP VISUALIZE-IVL Documentation

HP 9000 Series 700 Computers

HP Part No. B5182-96001
Printed in USA E0496

Edition 1

FINAL TRIM SIZE : 7.5 in x 9.0 in

Notices

The information contained in this document is subject to change without notice.

Hewlett-Packard provides the following material "as is" and makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages (including lost profits) in connection with the furnishing, performance, or use of this material whether based on warranty, contract, or other legal theory.

Some states do not allow the exclusion of implied warranties or the limitation or exclusion of liability for incidental or consequential damages, so the above limitation and exclusions may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

Warranty. A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

"OpenGL" is a trademark of Silicon Graphics, Inc.

FINAL TRIM SIZE : 7.5 in x 9.0 in

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

May 1996 ... Edition 1. This manual is valid for all HP 9000 Series 700 computers running the IVL software under HP-UX release 10.20.

0

Preface: About this Documentation

Manual Contents

For your convenience, the document you are reading, the *Image Visualization Library Implementation Guide*, exists in two forms: a web-browsable version and a paper version. The web-browsable version exists on the World-Wide Web at Access HP (URL=`http://www.hp.com`), as well as on your local file system in the directory `/opt/graphics/IVL/doc/Web`, which can be accessed from your web browser even *without* an Internet connection. The paper version contains the same information as the web version.

This document contains the following information:

- Chapter 0—Preface: About this Documentation
  Contains information about the audience, formatting conventions, and contents of the IVL documentation. It also contains pointers to other sources for imaging information, directions for printing the IVL documentation, and information about recommended Web browsers.
- Chapter 1—For System Administrators
  Contains system administration tasks and information for the IVL product. This includes installation and configuration information, as well as information on compatible software revisions of IVL, HP-UX, and X11.
- Chapter 2—Overview of the Image Visualization Library (IVL)
  Contains a high-level overview and definition of the IVL API, with diagrams to describe hardware and software architecture.
- Chapter 3—For Application Developers
  Contains information you will need to develop IVL applications. This includes naming conventions, linking and compiling, supported data formats, tuning tips, and troubleshooting information.
- Chapter 4—Interaction with the X Window System
  Contains X-specific information you will need to develop an IVL application.

- Chapter 5—IVL Implementation and Device-Specific Information
  Contains device-specific information for the graphics devices supported by IVL.
- Appendix A—IVL Quick Reference
  Contains names and parameter lists of the IVL routines.
- Appendix B—IVL Reference
  Contains complete reference pages for each the IVL routine.
- Glossary
  Contains a list of IVL-related terms and their definitions.

Audience and Scope

This manual is designed to teach imaging application developers about the Image Visualization Library (**IVL**). This manual is not designed to teach you to become an application developer, or to teach you about imaging terminology and concepts. To use IVL, you will need knowledge of the following:

- Using HP-UX.
- C programming, linking, and compiling.
- X11 and Motif programming.
- Digital image processing terms and concepts.

Formatting Conventions

The IVL documentation uses the following formatting conventions:

- `Typewriter text` represents computer literals. This text should be typed in exactly as it appears.
- *Italic text* represents variable names. You should substitute your own text in place of the italics.
- **Bold text** represents `glossary` terms. See the Glossary for definitions.
- ⟨*Angle-bracketed italics*⟩ represents conceptual variables. These are not literals, but should be replaced by whatever value is appropriate for the context. For example, the following line:

  `cc` ⟨*filename* ⟩ `.c -lIVL -lm -o` ⟨*filename*⟩

  means you should type the above line exactly, replacing the two occurrences of ⟨*filename*⟩ with the name of an IVL program.

For More Information

Hewlett-Packard does not attempt to discuss in detail the concepts and theory of digital image processing in this manual. For further information on digital image processing, the following references may be helpful. Note that these are suggestions, a starting point for further reading, not a specific endorsement of these books over others not listed here.

- *Image Visualization Library Reference*—a reference section in this document that provides descriptions of IVL API routines, their use, and parameters.
- *Graphics Administration Guide*—an HP document that provides device support, pathname, and other information that is applicable to all of Hewlett-Packard's graphics APIs.
- *Digital Image Processing* (3rd Edition) by Rafael C. Gonzalez and Richard E. Woods (Reading, MA: Addison-Wesley; 1992). ISBN: 0-201-50803-6.
- *Digital Image Processing* (2nd Edition) by William K. Pratt (New York: John Wiley and Sons; 1991). ISBN: 0-471-85766-1.
- *Digital Image Warping* by George Wolberg (Los Alamitos, CA: IEEE Computer Society Press; 1990). ISBN: 0-8186-8944-7.

Viewing IVL Documentation with Web Browsers

At the time of publication for this document, Hewlett-Packard does not distribute a World-Wide Web browser with HP-UX. Nor does HP endorse or recommend any specific browser from other sources.

Informal testing has found some problems with early versions of various browsers. Because of this, the following versions of Web browsers are recommended for viewing the IVL documentation:

- **Netscape Navigator**^{TM}, version 1.1N or later. (At the time of publication, version 2.0 was the most recent supported version of Navigator.) This is available via anonymous FTP from `ftp2.netscape.com` (or `ftp3` or `ftp4` or `ftp5 ...`) in the directory `2.0/unix`; the file name is `netscape-v20-export.hppa1.1-hp-hpux.tar.Z`.
- **NCSA Mosaic** ^{TM} , version 2.5 or later. (At the time of publication, version 2.6 was the most recent supported version of Mosaic.) This browser is available via anonymous FTP from `ftp.ncsa.uiuc.edu` in the directory `/Mosaic/Unix/binaries/2.6`; the file name of the most recent version (as of this writing) is `Mosaic-hp-2.6.Z`.

Printing the IVL Documentation

The IVL product also includes a PCL file that you can use to print a paper copy of the IVL documentation. Printing these files requires a printer with PCL capabilities, referred to as ⟨*printer_name*⟩ below.

To print the *Image Visualization Library Implementation Guide*:

```
cd /opt/graphics/IVL/doc/printfiles
lp -d⟨printer_name⟩ -oraw ImplementationGuide.pcl
```

Please note that there are limitations in HTML capabilities and different capabilities among Web browsers. This means that there are likely to be differences between the appearance of IVL documentation when viewed with a browser and the printed version of the same documentation.

Contents

0. Preface: About this Documentation

Manual Contents 0-1
Audience and Scope 0-2
Formatting Conventions 0-2
For More Information 0-3
Viewing IVL Documentation with Web Browsers 0-4
Printing the IVL Documentation 0-4

1. Chapter 1: For System Administrators

Installation 1-1
IVL Filesets 1-1
Using SD-UX 1-1
X Configuration 1-2
X11 1-2
Double-Buffering 1-2
Single Logical Screen 1-2
VUE and CDE 1-3
Motif 1-3
Revision Information 1-4
Using the `what` Command 1-4
Using the `uname` Command 1-4
If You Have Incompatible Software 1-4
Using the `graphinfo` Command 1-6

2. Chapter 2: Overview of the Image Visualization Library (IVL)
What is IVL? 2-1
IVL Description 2-1
Relationship to OpenGL 2-2
Using IVL with Other Graphics APIs 2-2
Xlib and Motif 2-2
HP Image Library 2-3
Starbase, HP-PHIGS, and HP PEX 2-3
Other OpenGL Implementations 2-3
Color Model 2-4
Frame Buffer Organization 2-5
High-Level IVL Overview 2-7
Window Coordinate System 2-7
Rendering Contexts 2-8
Hardware and Software Architecture 2-9
The IVL Machine 2-10
Abstract Machine 2-10
Pipeline Stages 2-12
Unpack Pixels 2-12
Pixel Transfer 2-16
Pixel Rasterization 2-16
Fragment Operations 2-16
Frame Buffer 2-17
More on Pixel Transfer 2-17
Convolution 2-18
Post-Convolution Scale and Bias 2-21
Image Transform 2-21
Post-Image Transform Color Table 2-22
Conversion to Frame Buffer Resolution 2-22
Pixel Rasterization 2-23
Fragment Operations 2-23
Pixel Ownership Test 2-23
Scissor Test 2-24

3. Chapter 3: For Application Developers
Naming Conventions 3-1
IVL Routines 3-1
Standard IVL Routines and Constants 3-1
Window System Routines and Constants 3-2
Extensions to IVL 3-2
Extensions Supported by Multiple Vendors 3-2
Extensions Supported by HP Only 3-2
Function Variants Based on Data Type 3-3
Naming Conventions Summary 3-4
IVL Routines 3-4
IVL Data Type Names 3-4
IVL Constants 3-5
Types of IVL API Routines 3-6
Setting and Querying Attributes 3-6
Imaging Operations 3-6
Window System Interaction 3-7
Compiling and Linking 3-8
IVL Data 3-8
Supported Data Formats 3-8
Pixel Unpacking 3-9
Implementation Restrictions 3-10
Underlays 3-10
Distributed Environments 3-10
Multi-Threaded Applications 3-10
Programming Advice 3-11
Query Data Values 3-11
Image Data Formatting 3-12
RGBA Data Format 3-12
Performance Tuning Tips 3-13
General Performance Hints 3-13
Performance Hints for Workstations with IVX Hardware . . . 3-13
Performance Hints for Workstations without IVX Hardware . 3-14
Error Handling 3-15
Sample Code 3-19

4. **Chapter 4: Interaction with the X Window System**
X Interaction 4-1
X Windows Capabilities 4-4
Setting Up an X/IVL Program 4-4
Selecting Visuals 4-5
IVL and Color Recovery 4-6
Managing Rendering Contexts 4-6
Double-Buffering Support 4-7
IVL and Backing Store 4-8
Using Pixmaps 4-8
Synchronization 4-8
Using the `glFinish` Routine 4-8
Using the `glFlush` Routine 4-9

5. **Chapter 5: IVL Implementation and Device-Specific Information**
List of Devices 5-1
IVL Implementations 5-1
Renderer Names 5-2
Entry-Level Color Graphics Devices 5-3
Device Description 5-3
Supported Visuals 5-3
Color Map Management 5-4
Overlay Transparency 5-4
HCRX Family Device Descriptions 5-5
Device Descriptions 5-6
HCRX-8 Description 5-6
HCRX-24 Description 5-6
Supported Visuals 5-7
HCRX-8 5-7
HCRX-24 5-7
Color Map Management 5-7
Changing the Default Visual 5-8
Overlay Transparency 5-8
Overlay Transparency with HCRX-8 Devices 5-8
Overlay Transparency with HCRX-24 Devices 5-9
Image Visualization Accelerator Device Description 5-10
Device Description 5-10
Angle of Rotation 5-10

Rasterization 5-10
Performance Hints 5-11
Clip Rectangles 5-11
Software versus Hardware-Accelerated Paths 5-11

A. Appendix A: Quick Syntax Summary for IVL
IVL Rendering Routines A-1
GLX Utility Routines A-3

B. Appendix B: HP-IVL Reference
`glClear` B-2
`glClearColor` B-4
`glColorTableEXT` B-5
`glColorTableParameter*vEXT` B-8
`glConvolutionFilter2DEXT` B-10
`glConvolutionParameter*EXT` B-13
`glCopyPixels` B-18
`glDrawBuffer` B-21
`glDrawPixels` B-23
`glEnable, glDisable` B-27
`glFinish` B-29
`glFlush` B-30
`glGet*v` B-31
`glGetColorTableEXT` B-36
`glGetColorTableParameter*vEXT` B-38
`glGetConvolutionFilterEXT` B-41
`glGetConvolutionParameter*vEXT` B-43
`glGetError` B-46
`glGetImageTransformParameter*vHP` B-48
`glGetString` B-50
`glImageTransformParameter*HP` B-52
`glIsEnabled` B-57
`glPixelStore*` B-59
`glPixelTransfer*` B-63
`glRasterPos*` B-66
`glReadBuffer` B-68
`glReadPixels` B-70
`glScissor` B-73

glX.Intro B-75
glXChooseVisual B-79
glXCreateContext B-82
glXCreateGLXPixmap B-84
glXDestroyContext B-86
glXDestroyGLXPixmap B-87
glXGetConfig B-88
glXMakeCurrent B-91
glXQueryExtension B-93
glXSwapBuffers B-94

Glossary

Index

Figures

2-1. Frame Buffer Organization 2-5
2-2. Window Coordinate System 2-7
2-3. Rendering Contexts 2-9
2-4. Hardware and Software Architecture 2-10
2-5. Abstract Machine 2-11
2-6. Pipeline Stages 2-12
2-7. Skipping Rows and Pixels 2-13
2-8. Unpacking Pixels 2-15
2-9. More on Pixel Transfer 2-18
3-1. Supported Data Formats 3-9
4-1. X Interaction 4-2

1

Chapter 1: For System Administrators

Installation

IVL Filesets

You must use SD-UX to install the following filesets in order to successfully develop or execute IVL applications:

- DDA-SHLIBS
- IVL-SHLIBS

You must use SD-UX to install the following filesets in order to successfully develop IVL applications:

- DDA-SHLIBS
- IVL-PRG
- IVL-SHLIBS
- IVL-WEBDOC (optional)
- IVL-HARDCOPY (optional)
- IVL-DEMO (optional)

Using SD-UX

See your HP-UX system administration documentation for information on using the Software Distributor on HP-UX.

X Configuration

X11

Double-Buffering

IVL draws images from the bottom to the top of a window. This is known as the **rasterization order**. (The OpenGL standard does not dictate a specific rasterization order, but does dictate that the image data coordinate system has it's origin at the lower left corner of an image. This coordinate system lends itself naturally to a bottom to top rasterization.) This differs from the **frame buffer**, which refreshes from the top to the bottom of the screen. Because of this, you may see a tearing effect as your image is being drawn. To hide this artifact of the rasterization order, you can use hardware double-buffering on those devices that support it (see `glXSwapBuffers`).

When you do this, you may still see some minor tearing due to buffer swapping. This is a less severe artifact than the one described above. You can eliminate this secondary tearing by forcing the X Server to swap hardware buffers during the vertical retrace interval. To do this, add the following to your X*screens file:

```
Screen /dev/crt
    ScreenOptions
        SwapBuffersOnVBlank
```

Note: since this affects hardware buffers only, it will have no effect on graphics systems that do not support hardware **double-buffering**. Also, synchronizing with vertical retrace may cause a slight decrease in performance.

Single Logical Screen

With Single Logical Screen (SLS), the X11 server manages multiple physical display devices as if they were a single frame buffer. Thus the use of the term "logical".

The initial release of IVL does not support the Single Logical Screen environment. This includes the case where IVL renders to a drawable that resides on a single physical display (frame buffer). To reiterate, SLS is *not* supported.

However, IVL *is* supported on multi-display configurations that are not configured as a Single Logical Screen.

VUE and CDE

IVL does not require any special configuration for VUE or CDE.

Motif

By default, Motif creates child widgets using the same visual class as their parent widget. This can cause your application windows to inherit an unexpected visual type. In order to support the rich set of visual classes available on HP workstations used in Motif applications, an alternative widget creation procedure is required.

A sample widget is provided for you to use. The directory `/opt/graphics/IVL/demo/DrawingA` contains the source files, header files, and a makefile. Incorporate the object file `drawinga.o` created by this set of files into your application.

Instead of making a procedure call to `XmCreateDrawingArea` to create a drawing area widget, applications should call `HPCreateVisualDrawingArea` using the same parameter list. It accepts an argument to specify a visual class different from a parent widget.

Applications can subsequently create OpenGL contexts using the same visual class used to create the drawing area widget. Call `glXMakeCurrent` to bind the context to the realized **drawable**.

See the example source file for details about Motif drawing area creation.

Revision Information

The following sections will help you determine what revisions of IVL, X11, and HP-UX you have on your system. Your operating system must be Release 10.10 HP-UX or a subsequent version, in order to use IVL. Refer to the *Graphics Administration Guide* to determine whether or not these revisions are compatible.

The output of the `what` command will show the name of the product, the version number, compile date, and the base operating system name (HP-UX) and it's release level.

Using the `what` Command

You can use the `what` command to make sure you have compatible revisions of IVL and X11 installed on your system.

To find the revision of IVL installed on your system, type:

```
what /opt/graphics/IVL/lib/libIVL.sl
```

To find the revision of X11 installed on your system, type:

```
what /usr/bin/X11/X
```

Using the `uname` Command

You can use the `uname` command to determine the revision of HP-UX installed on your system. To do this, type:

```
uname -r
```

For HP-UX revision 10.10 on Series 700 workstations, the output of the above command is `B.10.10`.

If You Have Incompatible Software

Once you have determined the IVL, HP-UX, and X revisions on your system, refer to the *Graphics Administration Guide* to determine whether or not these revisions are compatible.

If you have incompatible revisions of HP-UX, X11, and/or IVL installed on your system, you will need to update to the most recent and compatible revisions

of these software products. If you do not already have the necessary software, contact your local Hewlett-Packard sales office or your HP Response Center.

Using the `graphinfo` Command

The `graphinfo` command can be used to determine if you have image acceleration hardware installed on your system. To do this, type:

```
graphinfo | more
```

If you have Imaging Visualization Accelerator (**IVX**) hardware installed on your system, the following line will appear in the "`CONFIGURATION INFORMATION`" section:

```
image accelerator: yes
```

If you do not have IVX hardware installed on your system, the "`CONFIGURATION INFORMATION`" section will not include an "image accelerator" line.

On systems with IVX hardware, the entire section will look similar to the following output. (Note that this is a partial listing of what the `graphinfo` command reports for the HCRX-8 plus IVX.)

```
CONFIGURATION INFORMATION

    image planes:           8
    overlay planes:         8
    resolution:             1280 X 1024
    color or grayscale:     color
    PHIGS supported:        yes
    hardware accelerator:   no
    geometry accelerator:   no
    image accelerator:      yes
    texture accelerator:    no
    hardware zbuffer:       no
    software zbuffer:       yes
    video out:              no
```

2

Chapter 2: Overview of the Image Visualization Library (IVL)

What is IVL?

IVL Description

The Image Visualization Library (IVL) is an Application Programming Interface (**API**) from Hewlett-Packard that provides access to high-performance capabilities for the display and manipulation of two-dimensional images. IVL is a device-independent API; application developers do not need to provide special code to change between different devices. IVL will automatically take advantage of increased performance from the Image Visualization Accelerator (IVX) hardware if it is available. If the IVX hardware is not available, IVL provides the same capabilities through its software implementation.

IVL provides access to the frame buffer with the highest possible performance and the lowest possible overhead. IVL does not include elaborate image processing algorithms, nor does it support high-level abstractions for **image formats**. The API provides an efficient path for transferring pixels to the frame buffer. Because of its low-level focus, it is entirely appropriate to build toolkits and middleware products layered on IVL to provide optional utilities and functions that simplify application development.

IVL can be thought of as the server in a client/server model. An application (the client) issues commands, and these commands are interpreted and processed by IVL (the server). The client and server may or may not be running on the same processor. Because of this client/server model, IVL applications can operate successfully across a network or in a standalone environment. (Note: the initial release of IVL does not support the ability to operate across a network. Applications must run on the system where IVL is installed.)

The target customers for IVL are application developers working on image processing and display software. For example, an application developer creating

a diagnostic imaging application for the medical market would benefit from the high-performance image processing capabilities provided by IVL.

IVL currently supports **luminance format** (`GL_LUMINANCE`) and **RGBA format** (`GL_RGBA`) data, which are described in detail in the "IVL Data" section of the "For Application Developers" chapter. Future releases of IVL may support additional data formats.

Relationship to OpenGL

IVL is a library for image processing with a programming interface very similar to the imaging portions of the **OpenGL**® API and the defined imaging **extensions** to OpenGL. While the OpenGL API is not often thought of as an API for imaging, it was designed to expose the capabilities of modern frame buffer hardware. The emphasis in the OpenGL API is on 3D graphics, but it also includes a fairly rich set of capabilities for 2D image processing. The core capabilities of the OpenGL API can be extended using imaging proposals from Silicon Graphics, Inc. and others.

To the extent that IVL utilizes the OpenGL command syntax and state machine, it is used with permission from Silicon Graphics, Inc. However, HP makes no claims that IVL is in any way a compatible replacement for the OpenGL interface or associated with Silicon Graphics, Inc.

IVL is a stand-alone library that implements the imaging portions of the OpenGL API and some of the **OpenGL imaging extensions**. Because of its similarity to the OpenGL API, software written using IVL can be easily ported to an OpenGL environment. IVL provides a small, well-defined set of capabilities for pixel processing. The IVL entry points are identical in syntax and semantics to their counterparts in OpenGL. The only difference is that IVL is not a complete OpenGL programming environment.

Using IVL with Other Graphics APIs

Xlib and Motif

Application developers can mix calls to IVL, Xlib, and Motif in the same program. It is the application developer's responsibility to call various API synchronization routines to ensure that rendering occurs in the desired order.

HP Image Library

Using IVL in the same application with the HP Imaging Library is not supported. You can use both IVL and the HP Image API within the same application. (The Image API manipulates image data, but does not display it. You must use Xlib to display images after processing by the Image API.)

Starbase, HP-PHIGS, and HP PEX

Using IVL in the same application with a 3D API such as Starbase, HP-PHIGS, or HP PEX is not supported. There is no method to ensure synchronization of display output between IVL and any of these other APIs.

Other OpenGL Implementations

Using IVL in the same application with other implementations of OpenGL is not supported. There would be name space conflicts at link time that could easily result in altering the behavior of IVL entry points.

Color Model

The set of rules for manipulating color values in a processing system is sometimes called a **color model**. IVL supports a color model that is based on processing pixel values with red, green, blue, and alpha values, hence it is called RGBA mode. The color model is determined by characteristics of the window in which drawing is to occur. For this release of IVL, all windows that support IVL rendering support RGBA mode.

RGBA mode is based on the premise that the system supports the processing of up to four channels of color information simultaneously. The red, green, and blue components are always treated identically.

During processing, red, green, blue, and alpha values are conceptually treated as floating-point numbers in the range [0.0, 1.0]. As pixels are processed and converted into values that can be written into the frame buffer, a component value of 0.0 will be mapped into the smallest displayable frame buffer value, and a value of 1.0 will be mapped into the largest displayable frame buffer value.

In most cases, alpha values are treated the same as the other three components, but there are some differences in how alpha values are processed. However, the IVL specification permits implementations to streamline internal processing so long as doing so does not alter the resultant image. If the underlying frame buffer does not support the storage of an **alpha channel**, then the implementation may choose not to apply some image processing operations to the alpha data.

Many imaging applications manipulate and display images that contain only a single channel of color information. In IVL terminology, these images are referred to as luminance-only images, or simply **luminance** images. These luminance images can be thought of as RGBA images where the input luminance value is used as the red, green, and blue component value for each pixel. The alpha value defaults to 1.0 for every pixel.

Frame Buffer Organization

In IVL, the frame buffer is a two-dimensional memory array that holds pixel values. A portion of the frame buffer is typically visible on the display screen. Some portions of the frame buffer may never be visible.

Corresponding bits from each pixel in the frame buffer are considered to be a **bitplane**; each bitplane consists of a single bit from each pixel in the frame buffer. Bitplanes are grouped into **logical buffers**. The only logical buffer that is supported in this release of IVL is the **color buffer**.

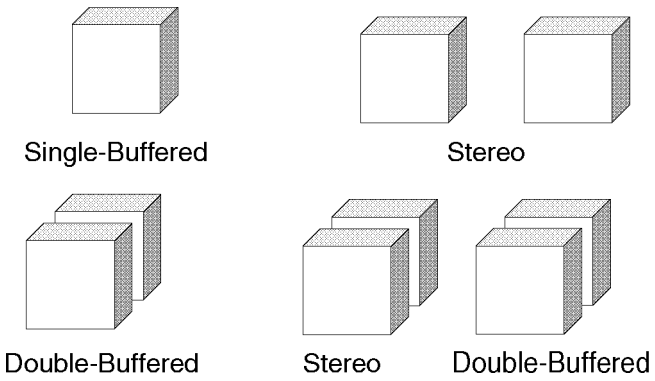

Figure 2-1. Frame Buffer Organization

The color buffer may consist of a number of buffers depending on whether it is single-buffered, double-buffered, stereo, or stereo double-buffered. The components of a color buffer are therefore referred to as the **front buffer** and the **back buffer**. For stereo frame buffers, the terminology is front left buffer, front right buffer, back left buffer, and back right buffer. Monoscopic frame buffers, by definition, contain only the left-side buffers.

The **current draw buffer** is the buffer that is the target of all subsequent rendering operations. You can set the current draw buffer using `glDrawBuffer`.

There is slightly different default behavior depending on whether the window is single-buffered or double-buffered. In order to make your application work in either type of window, you should be aware of this difference. If the window is single-buffered, the default **draw buffer** is `GL_FRONT`. If the window is double-buffered, the default draw buffer is `GL_BACK`.

If your application is inherently monoscopic, you should use the tokens GL_FRONT and GL_BACK when setting the current draw buffer. On non-stereo double-buffered windows, there are just two buffers, so these two tokens can be used to refer to each one explicitly. On a stereo double-buffered window, the token GL_FRONT will cause drawing to occur in both the front left and the front right buffers of the stereo window. Similarly, GL_BACK will cause drawing to occur in both the back left and back right buffers of the stereo window. Using the tokens GL_FRONT and GL_BACK will allow your monoscopic application to run properly on either **monoscopic windows** or **stereoscopic windows**.

This release of IVL does not support stereo windows, but the interface provides a migration path for applications to eventually include stereo display output. If you plan to eventually make your application stereo-capable, you should use the more explicit tokens GL_FRONT_LEFT and GL_BACK_LEFT. On non-stereo double-buffered windows, these tokens refer explicitly to the front and back buffers. On stereo double-buffered windows, these tokens refer specifically to the left-side buffers; the right-side buffers are not affected by subsequent drawing operations. In order to support stereo viewing, you will have to add code later to render the right side of the stereo image into the right-side buffers, and you will need branches in your program to skip right-side rendering when drawing in a monoscopic window.

The buffer from which pixels will be obtained during pixel read operations is known as the **current read buffer**. You can set the current **read buffer** using glReadBuffer.

The tokens GL_FRONT and GL_BACK have slightly different meanings with glReadBuffer than they do with glDrawBuffer.

In the context of the glReadBuffer routine, the token GL_FRONT refers specifically to the front left buffer and GL_BACK refers specifically to the back left buffer. (For pixel reading operations, it would make little sense to have GL_FRONT refer to both the front left and front right buffers. You only want to read pixels from one buffer at a time.) So for this routine, GL_FRONT and GL_FRONT_LEFT are synonymous, as are GL_BACK and GL_BACK_LEFT. If you are dealing with stereo windows and you need to differentiate the left and right buffer, you should use the more explicit terms GL_BACK_LEFT and GL_FRONT_LEFT.

High-Level IVL Overview

Window Coordinate System

Some IVL routines require you to provide locations in window coordinates. The window coordinate system in IVL has its origin (0,0) in the lower left corner of the window. Both x and y coordinates may be negative, and they may be larger than the window's *width* and *height*, respectively. But the visible pixels have coordinates from 0 to *width*−1 in the horizontal direction and 0 to *height*−1 in the vertical direction.

Another thing to keep in mind is that the IVL coordinate system has pixels that are centered on half-integer coordinates. In other words, if you draw a pixel at (0, 0), the pixel center will actually be at (0.5, 0.5). This is important when discussing **clipping** boundaries and precise positioning of images. The following figure shows a 3×5 rectangle whose lower left corner is at the window coordinate location (0, 0).

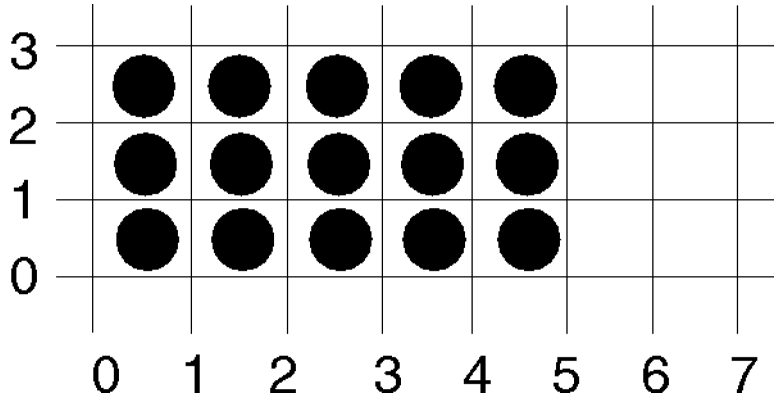

Figure 2-2. Window Coordinate System

Rendering Contexts

IVL is an API for a state machine. This state machine operates according to a very specific set of rules. Its behavior is deterministic: given the current values of all state attributes and a specific input, you can very accurately predict the output. By manipulating the state, the behavior of the underlying system can be modified.

There are two types of state in IVL. The first type of state is **server state**. Server state resides in the server and controls the rendering process. The mechanism for encapsulating server state information is called the **rendering context**. The majority of IVL state is stored in the rendering context.

The second type of state resides in the client and is called **client state**. The main purpose for this state info is to support remote rendering. The client state is maintained within the application data space.

Each instance of a rendering context implies one complete set of server state. Each connection from a client to a server implies one complete set of client state and one complete set of server state.

This is not required, but is good programming practice. (Each instance of the server state data structure is fairly large, on the order of several kilobytes of data.)

In many ways, the IVL rendering context is very similar to the concept of a Graphics Context (GC) in the X environment. The following figure shows the IVL rendering model in the X environment. The drawable can be considered the "canvas" on which drawing occurs. X is capable of rendering simple 2D graphics and text, and the state values that are stored in the GC determine the behavior of the X rendering "crayon." Similarly, you can think of IVL as a separate renderer with different capabilities that can render into the same drawables as X. The IVL rendering context is what determines the behavior of the IVL rendering "crayon".

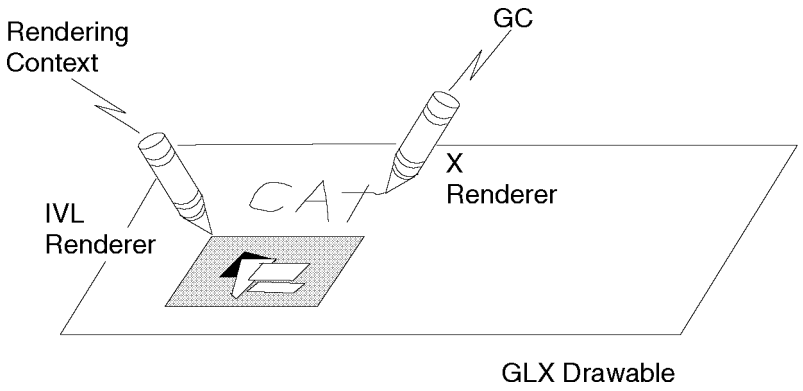

Figure 2-3. Rendering Contexts

To develop an IVL application in the X environment, you must first create a rendering context for a specific type of X visual. The `glXCreateContext` routine does this.

When a rendering context is no longer needed, deallocate it by calling `glXDestroyContext`.

Hardware and Software Architecture

The following diagram shows the relationship between IVL-related software and hardware. The diagram shows X-specific software on the left, IVL-specific software on the right, and display hardware in the shaded boxes.

When IVX hardware is available, IVL automatically uses its hardware implementation to write to the graphics frame buffer. When using a system without IVX hardware, IVL uses its software implementation to render an image. In either case, IVL uses the Direct Drawable Access Library (DDAlib) to access the hardware (frame buffer or accelerator).

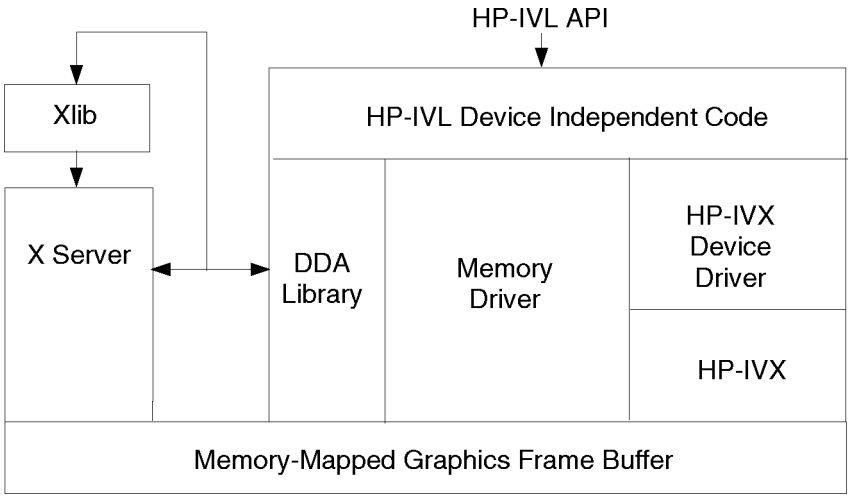

Figure 2-4. Hardware and Software Architecture

The IVL Machine

Abstract Machine

IVL is different from other low-level APIs in that it provides more than just a mechanism for moving pixels from one location to another; it actually defines a pixel-processing pipeline that operates on pixels as they are transferred. The following figure contains a diagram of the IVL state machine. This diagram shows how pixel data moves around the system and operations that affect pixels during transfer.

In this diagram, the arrows represent the flow of data within the system. Words that begin with "gl" indicate IVL subroutines that provide state or input values for that particular operation. Boxes that are outlined with thin lines and have square corners are operations that are applied to pixel values. Boxes with bold outlines and rounded corners represent storage locations for pixel values.

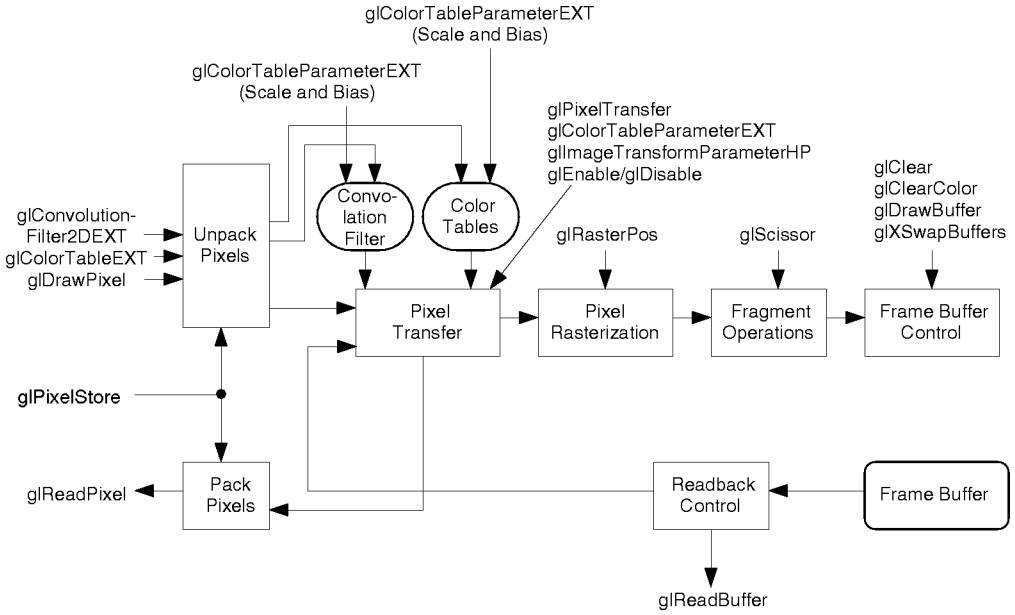

Figure 2-5. Abstract Machine

On the left side of the above diagram you'll see the three IVL routines that provide pixel information to IVL: `glConvolutionFilter2DEXT`, `glColorTableEXT`, and `glDrawPixels`. All three of these routines pass in a pointer to pixel values stored in host memory.

Pixel values in host memory can be stored in a variety of formats. The `glPixelStore` routines provide IVL with the parameters necessary to extract the desired pixel values. Using these **pixel unpacking** values, each of the three routines obtains the indicated pixels from host memory and sends them to their destination.

In the case of `glConvolutionFilter2DEXT`, the pixel values are modified by the scale and bias values set by `glConvolutionParameterEXT` and then stored in convolution filter memory for later use. Similarly, `glColorTableEXT` causes pixel values to be extracted from host memory, modified by the scale and bias values set by `glColorTableParameterEXT`, and stored in color table memory for later use.

Pipeline Stages

The ultimate destination for pixels specified by `glDrawPixels` is frame buffer memory. The pixel values follow a somewhat winding path to get to the frame buffer. The following figure is a more succinct diagram showing the steps that occur as part of the `glDrawPixels` routine. You should be able to trace a path through the state diagram in the previous section and see the same set of operations.

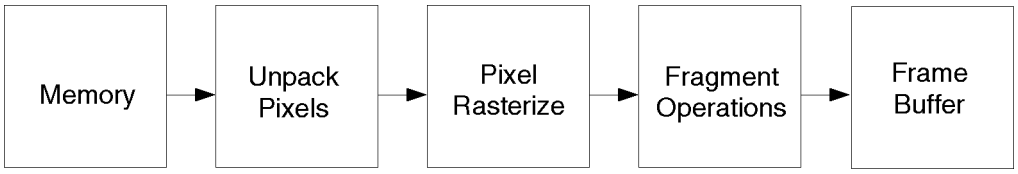

Figure 2-6. Pipeline Stages

The steps defined in the previous figure are described in the subsequent sections.

Unpack Pixels. For simple image transfers, the parameters for the `glDrawPixels` routine provides all the flexibility that is needed. The image is stored as a rectangle in host memory and you define the width, height, type, and format, along with a pointer to the start of the image data.

There are often times when additional flexibility is required. It is not uncommon for applications to maintain a large image (like 2K×2K) and transfer smaller portions of that image to the screen for display. The `glPixelStore` routine lets you transfer a **subimage**, and allows you to skip over padding at the end of each row of pixels.

A **pixel rectangle** is a two-dimensional array of pixels. Each pixel may contain one or four components depending on the *format* argument to `glDrawPixels`. Each pixel component has the **machine data type** specified by the *type* argument

to `glDrawPixels`. The pixels in memory can be thought of as a rectangle that is arranged in a series of rows. If no scaling or rotation factors are in effect, the first pixel group in the first row will be displayed at the current **raster position** as the lower left corner of the image.

The `GL_UNPACK_ROW_LENGTH` attribute can override the number of pixels in each row. If `GL_UNPACK_ROW_LENGTH` is 0, the number of pixels in each row is assumed to equal the width parameter passed to `glDrawPixels`, otherwise the number of pixels in each row is `GL_UNPACK_ROW_LENGTH`.

This attribute is typically used to extract a subimage with rows that are shorter than the real image stored in memory. If `GL_UNPACK_ROW_LENGTH` is less than the width you specify to `glDrawPixels`, you may see some "striping" effects. Keep in mind that this attribute indicates the number of pixels in each row, not the number of bytes. Note that IVL does not restrict the input arguments for pixel store operations to prevent these effects.

You can also skip a number of rows before reading the first row, and skip a number of pixels in that row (and each subsequent row) prior to the first pixel that is to be transferred to the display. `GL_UNPACK_SKIP_ROWS` defines the number of rows to skip, and `GL_UNPACK_SKIP_PIXELS` defines the number of pixels to skip in each row.

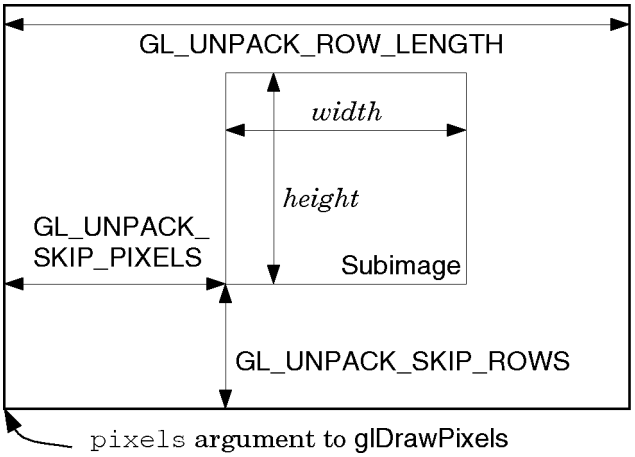

Figure 2-7. Skipping Rows and Pixels

The previous figure shows how these three pixel store values are used to extract a subimage from a larger image in host memory. The values indicated by *width*, *height*, and *pixels* are the values passed to the `glDrawPixels` routine.

To get to the first pixel that is to be transferred, the number of rows specified by `GL_UNPACK_SKIP_ROWS` is skipped. Each of these rows contains `GL_UNPACK_ROW_LENGTH` pixels. This positions the pointer at the beginning of the first row containing pixels to be transferred to the display.

Next, the number of pixels specified by `GL_UNPACK_SKIP_PIXELS` is skipped, positioning the pointer at the first pixel to be transferred. The next *width* pixels are transferred to the display. The pointer is then positioned at the start of the next row, and the process is repeated for each of the *height* rows that contain pixels to be transferred.

IVL also allows you to specify the data alignment per row via the `GL_UNPACK_ALIGNMENT` attribute. This attribute specifies whether each row begins at a memory address that is a multiple of 1, 2, 4, or 8 bytes. Set the alignment to 8 if each row begins at an address that is a multiple of 8, set it to 4 if each row begins at an address that is a multiple of 4, and so on.

The default value is 0 for the pixel store parameters that indicate row length, number of rows to skip, and number of pixels to skip in each row. The default value for alignment is 4.

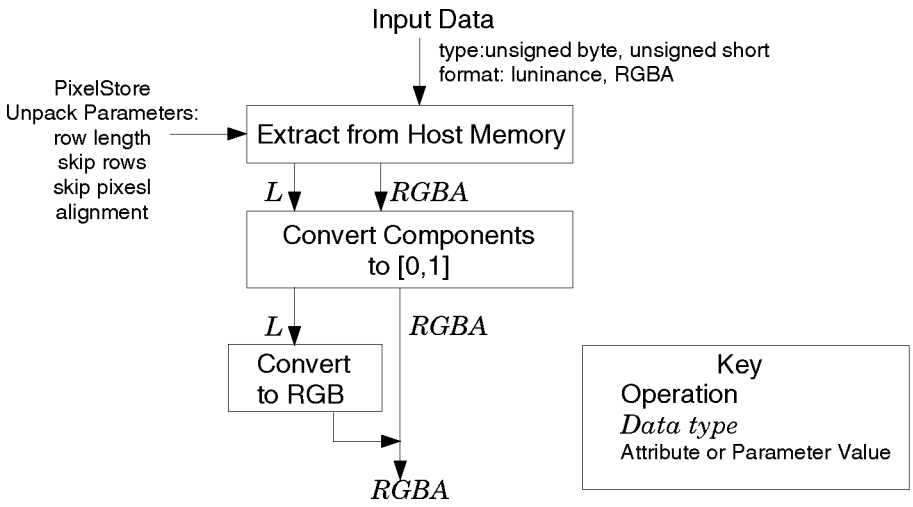

Figure 2-8. Unpacking Pixels

There are three main steps that go into unpacking pixels, as shown in the above figure. First, the arguments to `glDrawPixels` together with the pixel store parameters are used to extract pixels from host memory as described above. The pixels that are extracted will have components that are either unsigned bytes or unsigned shorts, and each pixel will consist of either one or four components, depending on whether the *format* argument is `GL_LUMINANCE` or `GL_RGBA`.

In order to simplify subsequent operations on these pixel values, the next conceptual step is to convert all **pixel components** to floating-point values in the range [0,1]. This conversion is shown as the second box in the above figure. (Note that this is a conceptual step in order to make it easier to specify the semantics of subsequent imaging operations. Real implementations will be optimized to avoid this conversion step if it is unnecessary). After completing this step, all pixel components are the same type and can be processed similarly.

The final conceptual step in unpacking pixels is to convert luminance values to RGBA values by assigning the luminance value to each of the red, green, and

blue pixel components and assigning a value of 1.0 to alpha. As the previous figure shows, the result of pixel unpacking is a stream of pixels that are uniform in type and format. This makes it easier to describe the operations that follow.

Pixel Transfer. After pixel values are extracted from host memory and unpacked, they undergo a set of operations collectively referred to as **pixel transfer**. Pixel transfer designates a set of operations that are applied whenever pixels are transferred from one place to another in the IVL environment (i.e., all draw, read, and copy pixel operations). This set of operations includes convolution, image transformation (scale, rotate, translate), and a look-up table operation.

As you can see from the state-machine diagram in the previous section, the pixel transfer operations are also applied when reading pixels from the frame buffer. It is important to realize that when you call `glReadPixels`, you must disable any pixel transfer operations that you do not want applied. (The pixel transfer operations also affect `glCopyPixels`, but the pixel copy path is not explicitly shown in the previous figure).

Pixel Rasterization. The next step in the image pipeline is **pixel rasterization**. This step primarily involves determining which frame buffer locations are to be modified as a result of the rendering operation. The routine `glRasterPos` is used to specify the window coordinate at which to place the resulting image. If the image is neither scaled nor rotated, the resulting image will be placed with its lower left corner at the specified position.

See the "Image Transform" section of this chapter for information on what happens if the image is scaled or rotated.

Rasterization produces a collection of fragments. A **fragment** consists of a color value and the coordinate of the frame buffer location at which that color value is to be written. Rasterization causes fragments to be generated for each frame buffer location that is affected by a call to `glDrawPixels`.

Fragment Operations. The IVL server applies various tests, collectively referred to as **fragment operations**, to fragments before they are written into the frame buffer. IVL currently defines two fragment operations: the **pixel ownership test** and the **scissor test**. The pixel ownership test determines whether each location that is to be written actually belongs to the current drawable. The scissor test discards pixels that fall outside of the rectangle defined by `glScissor`.

Frame Buffer. The pixel values that make it this far are written into the frame buffer. If the current drawable is double-buffered, the values may be written into the front or the back buffer, whichever is currently selected for drawing.

More on Pixel Transfer

The following figure is an expanded version of the "Pixel Transfer" box from the figure in the "Abstract Machine" section. Although the term "pixel transfer" is somewhat ambiguous, it refers to the set of operations illustrated in the following figure. These operations apply whenever pixels are drawn (`glDrawPixels`), read (`glReadPixels`), or copied (`glCopyPixels`). Most applications will use these operations only when drawing pixels (transferring pixels from host memory to the display). It is important to disable any of the capabilities that are not needed for pixel read and copy operations.

Clipping occurs during a later stage of processing. For the purposes of this discussion, it will be assumed that the resulting image lies completely within in the window or GLX pixmap. GLX pixmaps are defined in the "Using Pixmaps" section of the "Interaction with the X Window System" chapter.

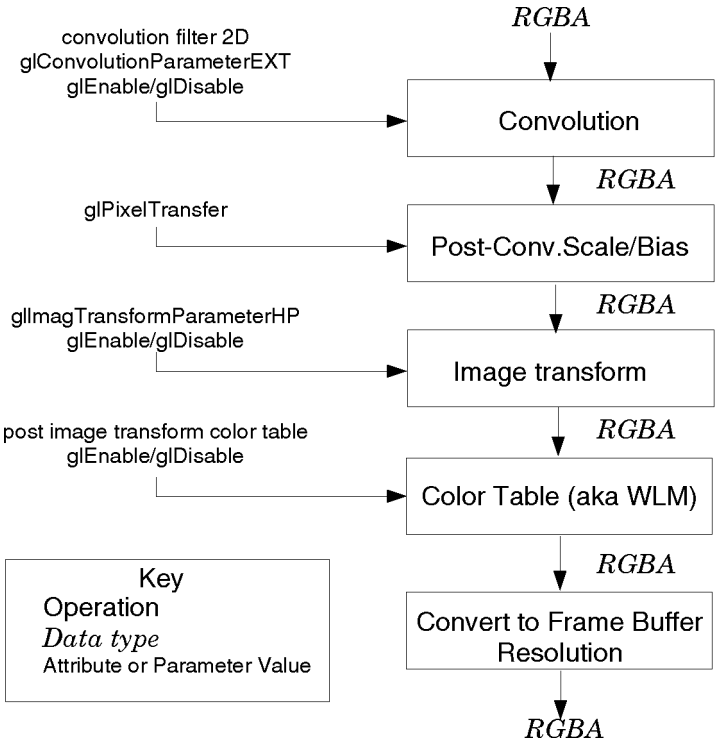

Figure 2-9. More on Pixel Transfer

The following sections expand on the concepts introduced in the previous figure.

Convolution

Convolution is a common image-processing operation used to filter an image. The filtering is accomplished by computing the sum of products between the source image and a smaller image called the **convolution filter** or **convolution kernel**. The convolution filter can be loaded with different values to achieve effects like sharpening, blurring, and edge detection.

If you want to perform a convolution operation as part of the pixel-processing path, the first thing you need to do is define the convolution filter. This can be done using the `glConvolutionFilter2DEXT` routine.

The pixels that make up the convolution filter pass through the "Unpack Pixels" stage of the pixel-processing path just as if `glDrawPixels` was called. However, instead of continuing on to the "Pixel Transfer" stage, the filter values are routed to the convolution filter memory for later use.

As the filter values are stored, they are modified by the current convolution filter scale and bias values. These attributes are set by `glConvolutionParameterEXT`.

These scale and bias values are provided in groups of four, one value for each of the red, green, blue, and alpha components. As each filter value is saved, its red component is multiplied by the red scale value and added to the red bias value, the green component is multiplied by the green scale value and added to the green bias value, etc. The resulting values are not clamped to the range [0,1] during this process. If *internalFormat* is set to `GL_RGBA`, each of the four resulting components is stored away in the convolution filter memory. If *internalFormat* is set to `GL_LUMINANCE`, only the resulting red values are stored away.

The convolution filter is a two-dimensional image indexed with coordinate (i, j) such that i increases from left to right, starting at zero, and j increases from bottom to top, also starting at zero. The convolution filter value at location (i, j) will be the Nth pixel, counting from zero, where $N = i + j \times filter\_width$.

Unless you really need to specify different convolution filter values for each of the red, green, blue, and alpha components, you should just specify an internal format of `GL_LUMINANCE`. This will result in a single component convolution filter that may provide slightly better performance than when using a four-component convolution filter.

The initial implementation of IVL currently supports convolution filters of size 3×3 only. The maximum supported values for the convolution filter width and height may be queried by calling `glGetConvolutionParameterivEXT` with the *pname* argument set to `GL_MAX_CONVOLUTION_WIDTH_EXT` or `GL_MAX_CONVOLUTION_HEIGHT_EXT`. It is a good idea to perform this query as part of your initialization routine so that you don't try to provide IVL with a convolution filter that is larger than it can handle.

The convolution operation is disabled by default and the default convolution filter is an empty filter. If you enable convolution without specifying a convolution

filter, the input image will pass through the convolution stage unmodified. You can query the current convolution filter using `glGetConvolutionFilterEXT`.

When it is enabled, the convolution operation occurs as part of the pixel-transfer operation and affects pixel rectangles that are transferred with calls to `glDrawPixels`, `glReadPixels`, and `glCopyPixels`. To enable the convolution operation, call `glEnable` with the value `GL_CONVOLUTION_2D_EXT`. You can disable convolution by passing the same value to `glDisable`, and you can see whether convolution is currently enabled by calling `glIsEnabled` with this value. In the default state, the convolution operation is disabled.

The convolution operation is a sum of products of pixels in the source image and pixels in the convolution filter. At this stage in the pixel-processing pipeline, source image pixels are always RGBA (four-component) pixels. If the convolution filter has an *internalFormat* of `GL_LUMINANCE`, it will be applied equally to red, green, and blue components of the source image, and alpha values will pass through unmodified. If the convolution filter has an *internalFormat* of `GL_RGBA`, the red components of the convolution filter will be convolved with the red components in the source image, the green components of the convolution filter will be convolved with the green components of the source image, the blue components of the convolution filter will be convolved with the blue components of the source image, and the alpha components of the convolution filter will be convolved with the alpha components of the source image.

When the `GL_CONVOLUTION_BORDER_MODE_EXT` attribute is set to `GL_REDUCE_EXT`, the sum of products is computed in the following fashion:

$$C(i,j) = \sum_{n=0}^{W_f-1} \sum_{m=0}^{H_f-1} C_{source}(i+n, j+m) \cdot C_{filter}(n,m)$$

In this equation:

- $C(i, j)$ is the result of the convolution operation for the output pixel with coordinates (i, j).
- $C_{source}(i, j)$ is the source pixel value with coordinates (i, j).
- $C_{filter}(n, m)$ is the convolution filter pixel value with coordinates (n, m).
- W_f is the width of the convolution filter and H_f is the height of the convolution filter.

When the **convolution border mode** is set to `GL_REDUCE_EXT`, the output image will have coordinates that range in i from 0 to the width of the source image

minus the width of the convolution filter, and that range in j from 0 to the height of the source image minus the height of the convolution filter.

See the `glConvolutionParameterEXT` reference page for information on other supported border modes.

Post-Convolution Scale and Bias

The next box shown in the Pixel Transfer figure is "Post-Convolution Scale and Bias." The filter values are not clamped when they are loaded. This means that you may specify values outside of the normal legal range (such as negative numbers). But this also means that you may cause an underflow or an overflow condition to occur. To alleviate this, IVL provides a way to specify scale and bias factors for each component that are applied once the convolution operation has been performed.

The post-convolution scale and bias factors can each be specified as a quadruple, with one value for each of the red, green, blue, and alpha components. These scale and bias values are only applied if convolution is enabled. After the convolution operation has been applied, all resulting red values are multiplied by the red post-convolution scale factor and added to the red post-convolution bias factor. Green, blue, and alpha components are treated similarly.

Once the scale and bias factors have been applied, resulting component values are clamped to the range [0, 1]. (Remember, at this stage of processing, all component values are treated as floating-point values with values normally in the range [0, 1].) The post-convolution scale and bias values are specified by calling the `glPixelTransfer` routine.

Image Transform

Another common imaging operation follows convolution in the pixel-processing pipeline. The image-transformation operation provides support for image scaling, rotation, and translation.

Like other operations in the imaging pipeline, the image transformation can be enabled by calling `glEnable` and disabled by calling `glDisable` using the value `GL_IMAGE_TRANSFORM_2D_HP`. By default, the image-transformation operation is disabled.

When enabled, the image-transformation operation uses the current set of image-transformation parameters to compute a new window coordinate for each incom-

ing pixel. All of these parameters can be set with `glImageTransformParameterHP`. You can query any of the current image-transform parameters using the `glGetImageTransformParameterHP` routine.

Post-Image Transform Color Table

Following the image-transformation stage of the pixel-processing path, it is possible to re-map all component values through the use of a **look-up table**. A **color table** uses each incoming pixel component value as an index into a table (that corresponds to that component). Thus, there are actually *four* look-up tables, one for each of the red, green, blue, and alpha channels. The value stored in the table at the indexed location is extracted and becomes the pixel component for subsequent processing.

The look-up table that immediately follows the image transformation stage of the pixel-processing pipeline is called the **post-image transform color table**. You can specify the contents of this table with `glColorTableEXT`.

Conversion to Frame Buffer Resolution

The final step of the pixel transfer stage involves converting pixel values from their internal floating-point representation into values that map to the entire range supported by the destination color buffer. First, pixel components are clamped to the range [0, 1]. Next, if the destination buffer has m bits, each component, c, is converted to the fixed point value k where $k=\{0, 1, \ldots, 2^m-1\}$ and the fraction $k/(2^m-1)$ is closer to c than any other value.

Another way of looking at this is with a concrete example. If your destination color buffer is 8 bits, then you have $2^8 = 256$ different values that are possible. The floating-point values in the range [0, 1] are mapped onto the 256 frame buffer values with 0.0 being mapped to frame buffer value 0 and 1.0 mapped to frame buffer value 255. The range [0, 1] is effectively divided into 255 bands, and any floating-point value in band k will get mapped to a frame buffer value of k.

Pixel Rasterization

Rasterization is the process of converting a rendering primitive into a collection of window-coordinate values and pixel values to be written at those locations. You can think of rasterization as consisting of two steps: first, determine all of the pixel locations in a window that will be affected by rendering the primitive, and then determine what value is to be written at each of the affected locations.

Rasterizing a pixel rectangle without any scaling or rotation is pretty straightforward: the pixel values in the input image have a one-to-one correspondence with pixels in the frame buffer. In the simplest case, rasterizing a pixel rectangle is simply a matter of copying it from host memory to frame buffer memory. More elaborate processing is required when the input image is scaled or rotated, or when the frame buffer organization is quite different from the format of the image in host memory.

The `glDrawPixels` routine does not have any arguments that specify where the resulting image is to be drawn. The location at which to draw a pixel rectangle is called the raster position. The raster position is stored as the current state and is specified with one of the `glRasterPos` routines. You can query the current raster position by calling `glGetIntegerv` with the *pname* argument set to `GL_CURRENT_RASTER_POSITION`. The default raster position is (0, 0).

Fragment Operations

All fragments that are generated by the rasterization process are subjected to additional pixel-by-pixel operations. Any fragments that make it through all of these operations without being discarded will be written into the current draw buffer. The two fragment operations defined in this release of IVL are the **pixel ownership test** and the **scissor test**.

Pixel Ownership Test

The first fragment operation that is applied is called the pixel ownership test. The pixel location at which the fragment is to be written is checked to see whether it is part of the current drawable. It may be that the pixel location to be written is obscured by another window, or the pixel location might be ineligible since it would be clipped by a window clip list.

If the pixel location that is to be written is not owned by the current drawable, the window system determines what to do with the incoming fragment. Most of the time, the fragment will be discarded. However, this test is defined in such a way as to allow window-system-specific behavior for conditions such as overlapping windows.

Scissor Test

IVL provides for a rectangular clipping region known as the **scissor box**. When the scissor test is enabled, the `glDrawPixels` routine can affect only the pixel values inside this rectangular region. When the scissor test is disabled, any of the pixels in the drawable can be modified. Scissor testing can be enabled or disabled by passing the value `GL_SCISSOR_TEST` to `glEnable` or `glDisable`. The `glScissor` routine may be used to set the current scissor box.

The scissor test is disabled by default. You can obtain the values for the current scissor box by calling `glGetIntegerv` with the *pname* parameter set to `GL_SCISSOR_BOX`. You can see whether the scissor test is enabled by calling `glIsEnabled` with the *cap* argument set to `GL_SCISSOR_TEST`.

3

Chapter 3: For Application Developers

Naming Conventions

Using IVL is easier if you remember some rules about the names used for functions, data types, and constants. For the purposes of writing portable code, it is also important to understand the naming conventions that are applied to extensions.

IVL Routines

Standard IVL Routines and Constants

In IVL, most procedure names begin with the prefix `gl`. (This is the naming convention for OpenGL, the "parent" of IVL.) Individual words within a procedure name begin with an uppercase letter (e.g., `glDrawPixels`).

Similarly, most data types that are defined as part of IVL begin with the prefix `GL`, with subsequent letters in lower case (e.g., `GLenum`). Underscores are not used in procedure names nor in data type names.

IVL includes pre-defined constants whose names begin with the prefix `GL_` and appear as all uppercase letters, with words separated by underscores (e.g., `GL_RED_BITS`).

Window System Routines and Constants

Some IVL routine names begin with the prefix `glX` (e.g., `glXMakeCurrent`). This prefix indicates that the routine is specific to the X Window System environment. Applications that require portability to windowing environments other than X should isolate calls to these routines in the window system dependent portions of their code.

Data types that are similarly specific to the X environment have a prefix of `GLX`, but follow the X convention of capitalizing the first letter of each subsequent word (e.g., `GLXContext`).

Extensions to IVL

There are a few exceptions to the rules mentioned above. Some capabilities in IVL derive from OpenGL extensions rather than the OpenGL API itself. These extensions are not currently part of the OpenGL standard.

In order to encourage cooperation and similarity among vendors developing and shipping OpenGL extensions, companies have agreed to identify their proprietary extensions with a company identifier (such as `HP`). All new procedure names, data types, constants, and extension names will be identified with this suffix.

Extensions Supported by Multiple Vendors. If two or more vendors agree to implement and ship the same extension, the company identifier can be replaced with the identifier `EXT`. These conventions allow application developers to easily determine:

- Which procedures, data types, and constants are standard.
- Which are "common" extensions.
- Which are proprietary to a single company.

IVL includes some procedures that derive from such "common" OpenGL extensions. Several of these extensions are specific to imaging. If these extensions are added to a future revision of the OpenGL standard, the `EXT` suffixes will be removed.

Extensions Supported by HP Only. IVL also contains a few procedures and constants that are specific to HP. The procedure names in this category contain the suffix `HP` and the constant names contain the suffix `_HP`.

Hewlett-Packard has kept the number of proprietary procedures and constants to a minimum, but there are requirements to provide certain critical features that aren't defined by the OpenGL standard or any existing OpenGL extension.

HP has offered these extensions to the OpenGL community and is encouraging other vendors to implement them. If any other vendor agrees to support them, the suffixes may change from `HP` to `EXT`. Furthermore, if the OpenGL Architecture Review Board integrates the HP extensions into a future version of the OpenGL standard, then the suffixes will be eliminated.

Function Variants Based on Data Type

Some procedures in IVL come in several variations, differing only in the data type of the arguments they accept. The key to understanding the difference in the routines is to look at the last few letters of the name (prior to any extension suffix like `EXT` or `HP`). The table below summarizes the combinations of letters that may appear at the end of procedure names:

Characters	C Data Type	Description of Data Type
`i`	`int`	Integer, passed by value
`iv`	`*int`	Pointer to array of `int`s
`f`	`float`	Float, passed by value
`fv`	`*float`	Pointer to array of `float`s

Naming Conventions Summary

IVL Routines

The following table summarizes the naming convention used with IVL routines:

Naming Convention	Example	Meaning
`gl*`	`glEnable`	Standard OpenGL routine
`glX*`	`glXGetConfig`	X Window System-specific routine
`*EXT`	`glColorTableEXT`	"Common" OpenGL Extension
`*HP`	`glImageTransformParameterHP`	HP-specific Extension

IVL Data Type Names

The following table summarizes the naming convention used with IVL data types:

Naming Convention	Example	Meaning
`GL*`	`GLint`	Standard OpenGL data type
`GLX*`	`GLXContext`	X Window System-specific data type

Note: IVL does not include any data-type extensions.

IVL Constants

The following table summarizes the naming convention used with IVL constants:

Naming Convention	Example	Meaning
GL_*	GL_BLUE_SCALE	Standard OpenGL constant
GLX_*	GLX_RGBA	X Window System-specific constant
*_EXT	GL_REDUCE_EXT	"Common" OpenGL extension constant
*_HP	GL_WRAP_BORDER_HP	HP-specific extension constant

Types of IVL API Routines

There are three basic types of routines in IVL:

- Routines that modify state (attributes).
- Routines that cause some action (operations).
- Routines that control window system operations.

Setting and Querying Attributes

The IVL API reflects an underlying state machine with attributes that can be modified to make it behave in different ways. State attributes in IVL are orthogonal to one another, meaning that setting one state attribute doesn't affect the setting or behavior of any other state attribute. For example, if you set certain parameters for the convolution operation, they have no effect on the parameters for the image transformation operation, nor do they modify the behavior of the image transformation operation.

In order to get IVL to behave in a particular way, you may have to set many state attributes. IVL provides facilities for querying any of the state values that you can set, so you are not required to keep track of the current attribute values. See the `glGet` and `glIsEnabled` reference pages for a list of the state values that can be queried.

There are default values for every state attribute in IVL. The default value usually reflects the most commonly used value for that attribute. For instance, there are a number of modes that can be enabled or disabled. Since the typical case for each mode is that it is disabled, most of these state attributes have a default value of "disabled".

Imaging Operations

Routines that perform tasks above and beyond the setting or querying of a state attribute are called operations. An example of a routine in this category is `glDrawPixels`, which causes a rectangular block of pixel values to be transferred from host memory to the frame buffer. The behavior of this transfer depends on the current settings of all of the state values that affect the pixel processing pipeline. The net result of this routine is (usually) that an image is displayed in a viewable window.

Operations (such as those that occur as a result of a call to `glDrawPixels`) have a well-defined set of semantics. By understanding the semantics of an operation, you can choose settings for attributes necessary to achieve the behavior you want. The pixel transfer operations in IVL occur in a sequence that is best explained using a pipeline model. The order of operations is very specific, as are the data formats for input to and output from each pipeline stage. See "The IVL Machine" section of the "Overview of the Image Visualization Library (IVL)" chapter for greater detail on the pixel transfer pipeline.

Window System Interaction

There are a number of routines in IVL that deal with coordination between the native window system and IVL rendering operations. In the HP workstation environment, the native window system is X. Therefore IVL contains routines for:

- Querying the capabilities of X visuals.
- Selecting a visual for rendering.
- Creating and manipulating data structures to store IVL state attributes.
- Performing double-buffering.
- Synchronizing between rendering with X and IVL.

It is important to realize that IVL can be implemented as a separate process from the X server. If it is implemented this way, it is possible to have IVL rendering and X window system rendering occurring simultaneously in the same window. In order to achieve the correct results when the order of rendering is important, applications must use the synchronization primitives that are provided in X and in IVL. For more information on these primitives, see the "Synchronization" section in the "Interaction with the X Window System" chapter.

Compiling and Linking

IVL supports only the C programming language. The following example shows the command line used to compile the IVL program ⟨*filename*⟩:

```
cc -I /opt/graphics/IVL/include ⟨filename⟩.c  \
  -L/opt/graphics/IVL/lib -lIVL -lX11 -lm -o ⟨filename⟩
```


IVL Data

Supported Data Formats

A component is the fundamental building block for a pixel value. For instance, the red, green, blue, and alpha components make up an RGBA pixel value. A pixel value may consist of either one or four components. The *format* argument to `glDrawPixels` and other routines specifies the number of pixel components.

Each component may be specified as one of several **image types**. IVL currently supports the following image types as arguments to `glDrawPixels`:

- `GL_UNSIGNED_SHORT`
- `GL_UNSIGNED_BYTE`

This means that you can provide luminance (single-component) pixel values as either 8-bit or 16-bit quantities and RGBA (four-component) pixel values as 8-bit quantities. (Note: IVL does not currently support the combination of *type* `GL_UNSIGNED_SHORT` and *format* `GL_RGBA`.)

IVL currently supports the following image data formats:

`GL_LUMINANCE`: This is a one-component data format that stores luminance values. The data can be in an 8-bit or 16-bit format.
`GL_RGBA`: This is a four-component data format that stores red, green, blue, and alpha values. In C language programs, declare this as a (one-dimensional) array of bytes so as to avoid byte-swapping problems with certain computer architectures.

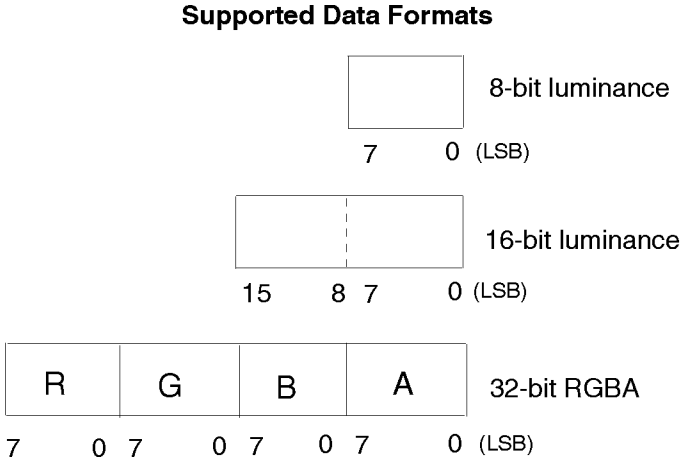

Figure 3-1. Supported Data Formats

Pixel Unpacking

Information on how IVL unpacks pixels is available in the "Unpack Pixels" section of the "Overview of the Image Visualization Library (IVL)" chapter.

Implementation Restrictions

Underlays

Hewlett-Packard graphics hardware does not support underlays, nor does IVL emulate this functionality in software.

Distributed Environments

HP does not currently support using IVL in a distributed (i.e., client/server) environment. Applications must run on the system where IVL is installed.

Multi-Threaded Applications

Use of IVL in a multi-threaded application is not currently supported.

Programming Advice

Query Data Values

Attributes are stored within IVL in the machine type that is used during processing. Some state attributes are stored as floats, some as integers, some as boolean values, etc. With some IVL routines, applications can specify a state attribute as either a float or an integer (for example, `glColorTableParameterfvEXT` and `glColorTableParameterivEXT`). In such cases, IVL converts and stores the passed value in the machine type that is used during processing.

It is considered good programming practice to query data values using the routine that corresponds to the data type of the value being queried. For instance, if you know you are querying an integer value, use the `glGetIntegerv` routine. Where necessary, the `glGet` routines will convert the requested value to the machine type specified by the routine. If the requested state attribute has a different type than is requested, it will be converted as follows:

- If a boolean value is to be returned, floating point and integer values are converted to `GL_FALSE` if and only if they have a value of zero. Otherwise, they are converted to `GL_TRUE`.
- If an integer is to be returned and the requested state attribute is a boolean, the returned value will be either `GL_TRUE` or `GL_FALSE`.
- If an integer is to be returned and the attribute is a floating point value, it will be rounded to the nearest integer unless it is a color (RGBA) value. In this case, the individual components will be mapped from their permissible floating point range of $[-1.0, 1.0]$ to the full range of allowable integer values (-1.0 maps to the largest representable negative integer, 0.0 maps to 0, 1.0 maps to the largest representable positive integer).
- If a float or a double is to be returned, boolean state attributes are returned as `GL_TRUE` or `GL_FALSE` and integer values are coerced to floating point values.

See the `glGet` reference page for a list of values that can be queried.

Image Data Formatting

If you are calling `glDrawPixels` with 12-bit image data stored as 16-bit values, you might be tempted to think that you only need to load a look-up table that contains 4096 entries. However, IVL has no way of knowing that only 12 bits out of the 16 are significant. If you create a look-up table with 4096 entries, IVL will (conceptually, at least) map each incoming 16-bit value to a float in the range [0,1]. The resulting value will then be multiplied by 4095 in order to compute the look-up table index. Thus, your incoming 12-bit input values will all be mapped into the first 256 entries of your color table.

One way to compensate for this effect would be to shift all your 12-bit values up by four bits. Then your 4096-entry color table would provide the expected results. Alternatively, you can load a full 64K-entry look-up table. If you go this route, you need to provide all 64K values, but you may only need to recompute the first 4096 entries, since the others might never be used. In practice you would be well-advised to load additional entries in the table since bicubic **resampling** in the image transformation stage might generate values that overflow 12 bits.

RGBA Data Format

Remember that `GL_RGBA` formatted data is stored in the following order: red, green, blue, then alpha. A common mistake is to format the data as alpha, red, green, then blue, which is incorrect.

Performance Tuning Tips

IVL is designed to take advantage of the acceleration provided by workstations equipped with IVX hardware. IVL contains support for rendering with IVX and support for rendering with a software implementation of the pixel-processing pipeline. The software implementation is used when no IVX accelerator is present, or when the limits of the IVX hardware are exceeded. Software rendering performance varies with the size of input data sets and the complexity of the display pipeline characteristics.

General Performance Hints

The following lists general hints for IVL performance improvements:

- IVL considers the lower-left corner of an image to be the origin. Images are rendered from the bottom row, up. To avoid the need for data conversion *by the application program*, store images in memory so the pixel at the lower-left corner is the first pixel in memory, followed by the remainder of the bottom row, then subsequent rows from the bottom to the top of the image.
- Angles of rotation will round to the nearest 1/10th degree. This is true for both the IVX and the software rendering paths.
- Do not arbitrarily intermix IVL and X procedure calls.
- For flicker-free, near-real-time, display of annotation, use the overlay visual and only change text that requires changing.
- Do not use `glXMakeCurrent` needlessly for every image frame. In other words, don't call `glXMakeCurrent` unless you are changing contexts.

Performance Hints for Workstations with IVX Hardware

Applications and end users should stay within the following limits to optimize performance on workstations using IVX hardware. Values that fall outside of these limits will operate correctly, but will use the slower software implementation.

- IVX supports X and Y scale values with absolute values between 0.75 and 32.0.
- IVX supports image translation values within the following limits:
  - $-8191.9 \leq x\_offset \leq 6911.9$
  - $-8191.9 \leq y\_offset \leq 7167.9$

- Image sizes should be at least 8×4 pixels (*width*×*height*), and at most 32768×32768 pixels.
- Only three clipping rectangles (including any enabled scissor specification) are supported in hardware. Additional clipping rectangles will cause rendering to be accomplished with multiple rendering passes through the hardware, so rendering performance may decease by more than half.

See the "IVL Implementation and Device-Specific Information" chapter for information about differences between using IVL's software implementation and using IVX.

Performance Hints for Workstations without IVX Hardware

Follow these hints to optimize performance on workstations that do not use IVX hardware:

- Nearest neighbor interpolation operates faster than bilinear interpolation, which in turn operates faster than bicubic interpolation. So, a **progressive refinement** scheme will provide the best results where interactivity is needed.
- Rotations that alter the x-axis direction will have slower performance than orientations that maintain a positive sense for x (such as no rotation, or a flip about the x axis using a negative y scalar). If your data is typically oriented to require rotating, consider reorienting the data instead of using image transformation to perform this operation.
- A scaling factor of 2.0 for both x and y (that is, 2× zoom) with bilinear resampling for luminance data performs better than any other image transformation.
- Avoid clearing a single buffer, and clearing and swapping double-buffers if not necessary.

Error Handling

IVL was designed to allow applications to achieve maximum performance. One such area is in the handling of error conditions. The error checking philosophy of IVL is fairly simple. IVL assumes that correctly working programs will pass valid parameter values. Exhaustive checking of all parameter values is usually only helpful when developing and debugging code.

Such error checking would adversely impact the performance of an application that was fully debugged and working properly. Therefore, primarily for performance reasons, IVL error checking is kept to a minimum. More exhaustive error checking can be performed by the application prior to calling IVL routines.

The goal of the IVL error semantics is to perform the minimum amount of error checking necessary to ensure that programs can continue to operate in a reasonable fashion (e.g., the program will not hang or crash the operating system). When an error is detected by IVL, the error value is recorded. The current error value is part of the current state and can be queried with the `glGetError` routine.

When an error occurs (indicated by setting the error value to something other than `GL_NO_ERROR`), no further errors are recorded until `glGetError` is called. The act of querying the current error value has the side effect of resetting the error value to `GL_NO_ERROR` in preparation for recording the next error.

If `glGetError` returns `GL_NO_ERROR`, then there have been no detectable errors since the last call to `glGetError`. The reference pages for each individual IVL routine describe errors that can be generated by that particular routine.

The following table describes the defined error values. The results of a routine are undefined only if the `GL_OUT_OF_MEMORY` error occurs. For all other errors, the offending routine is ignored and has no effect on the current state or the contents of the frame buffer.

If the routine that generates the error returns a value, it will return zero. If the routine that generates the error modifies values through a pointer argument, no change is made to these values. The fact that the offending routine is ignored can sometimes lead to mysterious program behavior. If your program seems to ignore some routines, insert calls to `glGetError` to ensure that no errors are occurring:

Offending Error Value	Description	Command Ignored?
`GL_INVALID_ENUM`	Enumerated value out of range	Yes
`GL_INVALID_VALUE`	Numeric value out of range	Yes
`GL_INVALID_OPERATION`	Operation illegal in current state	Yes
`GL_OUT_OF_MEMORY`	Not enough memory to execute command	Unknown
`GL_TABLE_TOO_LARGE_EXT`	Specified color table is too large to be stored	Yes

If IVL is implemented in a distributed (i.e., client/server) fashion, there may be more than one error value. To query and reset all the error values, you should call `glGetError` within a loop until a value of `GL_NO_ERROR` is returned. For each call to `glGetError`, one error value that is something other than `GL_NO_ERROR` will be returned and its value will be reset to `GL_NO_ERROR`.

It is a good idea to check the status of the current error value frequently in order to determine whether any errors have occurred. A good way to do this is to define a routine that checks for errors and then call this routine at the end of each rendering loop. For example:

```
{
     /* start of main rendering loop */
     /* ... */
     /* ... */
     /* end of main rendering loop */
     CheckForErrors ();
}
/************************************************************************/
static void CheckForErrors(void)
{
    int i;
    int gotError;

    gotError = FALSE;
    while (i = glGetError())
    {
        gotError = TRUE;
        fprintf(stderr, "Error: 0x%xn", i);
        process_error (i);
    }
    if (gotError)
        exit (1);
}
```

The disadvantage to this approach is that a number of pixel processing operations may occur before you check for errors. If this happens, it may be necessary to repeat the rendering operations after detecting and correcting the error.

Although the goal of IVL is to detect errors that would cause a fatal exception, there are some conditions under which a fatal exception might happen. If a sequence of floating point computations occurs (such as in the image transformation computation) with very large values (e.g., `MAX_FLOAT`), it is possible for an application to produce a floating point overflow exception.

Since such large data values are not likely to be produced by a properly working application, and since preventing this type of condition would be difficult and would penalize the performance of properly working applications, this condition may result in a floating point exception. It is the responsibility of the application

to provide values that are well within the range of floating point computation limits.

Sample Code

The IVL product includes some sample code to use as a learning tool. The `/opt/graphics/IVL/demo` directory contains files and data that demonstrate many of the features of IVL. There are smaller example programs in the `/opt/graphics/IVL/examples` directory.

`README` files in these directories describe how to make and run the programs. (Remember that the example code that ships with IVL is in a read-only part of the file system. You should copy those files elsewhere before modifying and/or compiling them.)

4

Chapter 4: Interaction with the X Window System

X Interaction

IVL is designed to be "window system neutral." That is, it attempts to avoid duplicating capabilities that are typically supported by the native window system (e.g., window operations, interactive input, color maps, and overlays). The focus of IVL is to provide 2D imaging capabilities. By not including these operations in the API, IVL can coexist with a variety of window systems.

However, the initial release of IVL is only available in the X Window System environment. In order for IVL to coexist peacefully with the native window system, a small number of window-system-dependent routines must be defined. These routines provide implementation information, configuration management, resource allocation/deallocation, synchronization, and other window system dependent functions.

In the X Window System environment, IVL is supported by an X server extension called **GLX**. This extension allows IVL to coordinate its rendering operations with those of X and other extensions. The names of routines in IVL that are specific to the X Window System environment are prefixed with "glX."

The following figure shows a block diagram of the X/IVL environment:

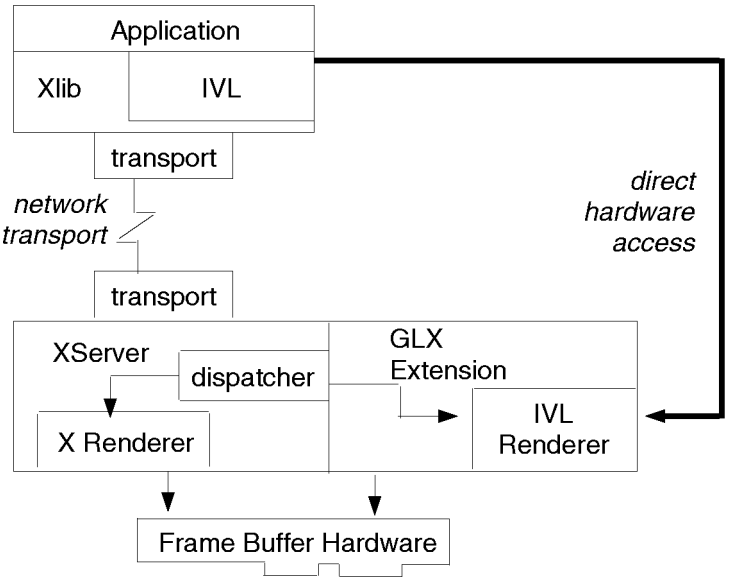

Figure 4-1. X Interaction

The top of the above figure shows the application making calls to both Xlib and IVL. These two libraries cooperate in sending commands to the X server via a network connection. The boxes labeled "transport" indicate the network transport mechanism that actually transfers X protocol requests between the X client (the application) and the X server.

A dispatcher inside of the X server decides whether the request should be handled by X itself or one of the available X extensions like GLX, which supports IVL rendering and other operations. If the incoming command is an X request, it is handled directly by the X server. Some of the X routines are used to perform simple graphics operations and will cause the X renderer to modify the contents of the frame buffer.

The biggest advantage of the X protocol is that it is network transparent. This makes it easy to develop an application running on one machine that communicates with an X server on another machine. However, the need to transfer commands via some network transport mechanism can be a performance

restriction if the application and the X server run on the same machine (as is often the case).

There are several ways that data can be transferred more rapidly between processes if the processes are known to be running on the same system. IVL provides a method for the application to directly access the frame buffer hardware, provided that the application and the X server are running on the same processor. Rather than going through the X protocol transport and the X dispatcher, calls to IVL are converted immediately into commands that directly access the frame buffer hardware. The improved performance that results from direct hardware access is shown symbolically by the width of the arrow (labeled "direct hardware access") connecting the IVL library and the IVL renderer on the right side of the previous figure.

Application developers should note that this release of IVL only supports local rendering. The ability for remote rendering with IVL across a network connection is under investigation by HP, and may be supported in a future IVL release. Direct (local) rendering (which is the highest-performance path for IVL rendering) is available and optimized in the initial release.

X Windows Capabilities

This section describes the steps involved in setting up an IVL program in the X environment. It then describes some of the IVL-specific steps in more detail. Finally, it describes other, optional capabilities of IVL in an X environment.

Setting Up an X/IVL Program

There are nine main steps for setting up an IVL application. They are listed below. (Note that it is good programming practice to check the return value from these functions and respond to any errors.)

1. **Open a connection**
   The first step is to call `XOpenDisplay` to establish a connection to the X server. This sets up the communication link to the X server.
2. **Check for the GLX extension**
   You can then call `glXQueryExtension` to ensure that the GLX extension is present.
3. **Select a visual type**
   Now select an appropriate **X visual**. The easiest method is to call `glXChooseVisual`, but you could also implement your own algorithm with calls to `XGetVisualInfo` and `glXGetConfig`.
4. **Create a window**
   Once you have decided which visual type to use, you need to create a window. One straightforward way of doing this is to set the necessary window attributes and call `XCreateWindow`. You may also want to set the standard properties for the window, such as the window name, the icon name, and so on.

   If you select a visual type that is different from the parent window's visual type and has a read/write color map, you should create the color map prior to creating the window and make sure that it is properly initialized.
5. **Create a rendering context**
   The next step is to call `glXCreateContext` to create a rendering context that matches the visual type of your window. The resulting rendering context will hold the state values necessary for rendering.
6. **Make the window and rendering context current**
   By calling `glXMakeCurrent`, you can establish both the current drawable and the current rendering context. This routine will generate an error if the visual type of the drawable does not match the visual type for which the rendering

context was created. Note: using `glXMakeCurrent` for each image frame will reduce application performance.

7. **Initialize rendering state**
   If you need to set any IVL state values to something other than their default values, you would probably want to call an initialization routine prior to falling into your main event processing loop. IVL state attributes are designed to have reasonable defaults, so you probably will not need to initialize too many values.
8. **Map the window**
   This step finally makes the window visible. Because of the previous steps, the window is ready for IVL rendering as soon as you call `XMapWindow`. There is a finite delay between the time you issue the map window command and the time the window appears on the screen. If you issue rendering commands during this period, they will not show up on the window when it becomes visible. Consequently, it is usually wise to wait until after the first `Expose` event, indicating that the window is actually visible on the screen.
9. **Main event loop**
   At this point, your X program will typically enter an event loop that waits for events (for example: expose, input, resize) to occur and then processes them. The `XNextEvent` call is the "wait for event" operation.

See the sample code provided with IVL for source code that demonstrates these steps (though not necessarily in the exact order shown above).

Selecting Visuals

Over the years, a number of frame-buffer architectures with different capabilities have been developed:

- Some store three channels of data (red, green, and blue), others store only one.
- Some allow pixel values to be routed through hardware lookup tables (color maps) in order to provide a level of indirection.
- Some are inherently monochrome, others are polychrome.
- Some are single layer, others are multi-layer (with overlay or underlay planes).

To expose these different capabilities in a portable way, the X Window System introduced the concept of a **visual type**. There are six visual types in X: `TrueColor`, `DirectColor`, `PseudoColor`, `StaticColor`, `GrayScale`, and `StaticGray`.

In the X environment, applications create windows (regions of the frame buffer that they can draw into) of a particular visual type. The visual type of a window defines its behavior in some important ways, and it is through this mechanism that frame buffer features are exposed and software portability is defined. To be specific, an `XVisualInfo` structure in X defines the visual type, screen, and depth of the window, as well as the number of bits of red, green, and blue information in a pixel. To achieve true portability, applications should work on systems that support any visual types that meet their minimum requirements.

In the X Window System environment, IVL adds attributes to the X visual type. These attributes can be queried using routines defined by the GLX extension which provides support for IVL in the X environment. All visual types that are supported are defined within the X server and are reported back to the client at connection time. Since X does not return information about the extended attributes defined by IVL, it is quite possible that an X server will report two or more visuals that look identical to the client. However, these visual types will differ in the IVL attributes they support, and the differences may be ascertained using the IVL routines provided for this purpose.

The easiest method to find a visual that meets your application's requirements is to use `glXChooseVisual`. This routine has its own prioritization algorithm to select the "best" visual from the list of attributes that you specify. If you wish to implement your own prioritization algorithm, use `XGetVisualInfo` and `glXGetConfig`.

IVL and Color Recovery

IVL does not support Color Recovery.

Managing Rendering Contexts

In order to render anything with IVL, you need to specify both a current rendering context and a current drawable (either a window or a **GLX pixmap**). The `glXMakeCurrent` routine accomplishes this. This routine seems a bit different from most other IVL routines in that it requires you to specify two separate pieces of state simultaneously.

The reason this routine requires you to set the drawable and the rendering context simultaneously is that the two are related. When you create a rendering context, you must specify a visual type. This tells IVL that the rendering context

will be used in conjunction with drawables of that type. The `glXMakeCurrent` routine checks to ensure that the rendering context you specify will work with the drawable you specify. If the drawable is not a visual type that will work with the rendering context you provide, IVL generates an error.

With `glXMakeCurrent`, it is easy to change both the drawable and the rendering context at the same time. This is something that applications drawing into multiple windows may need to do quite often. If IVL required you to change the current rendering context and the current drawable with independent commands, there would be no way to change the current rendering context and the current drawable for use on a different visual type without generating an error.

Double-Buffering Support

In order to eliminate the flicker effect caused by clearing and redrawing an image in a visible window, IVL supports the notion of double-buffering. Visuals that support double-buffering have two drawing buffers: a front buffer and a back buffer. The front buffer is displayed while rendering is occurring in the back buffer. When rendering is completed, use `glXSwapBuffers` to display the rendered image. Also, remember to set the current draw buffer using `glDrawBuffer`.

There are three supported ways for IVL applications to perform double-buffering:

- Using the double-buffering capabilities provided with IVL. See the `glXSwapBuffers` reference page for more information.
- Using the Double-Buffering Extension to X (DBE). See the documentation in the file `/usr/lib/X11/Xserver/info/screens/hp` for more information.
- Using the Multi-Buffering Extension to X (MBX). See the documentation in the file `/usr/lib/X11/Xserver/info/screens/hp` for more information. Also, please see the note below regarding use of MBX.

Please note that although MBX is supported with this release of IVL, it is not recommended. MBX has not been adopted as an industry standard, and it may become obsolete in the future. If you are currently using MBX, you should plan to migrate to one of the other double-buffering methods.

Furthermore, note that the combination of `glXSwapBuffers` and DBE in the same application is not supported; you must choose one or the other.

IVL and Backing Store

IVL does not support rendering into backing store.

Using Pixmaps

It is also possible to use IVL to render into off-screen memory. X **pixmaps** are the basis for supporting off-screen rendering. X is limited in that it does not define visual types for pixmaps. In X, pixmaps are not necessarily associated with a screen; they have a depth, but no visual type.

Since IVL extends the notion of visual types by defining additional visual attributes, the definition of a pixmap is also extended. IVL does this by defining a superset of an X pixmap, called a **GLX pixmap**. Unlike an X pixmap, a GLX pixmap has a visual type, so it may contain the extended visual attributes defined by IVL. In order to use a pixmap as a rendering destination, it is necessary to enable it for rendering with IVL by calling `glXCreateGLXPixmap`.

Once a GLX pixmap is no longer needed, deallocate it using `glXDestroyGLX-Pixmap`.

Synchronization

Using the `glFinish` Routine

For efficiency reasons, IVL contains the notion of a processing pipeline. For the most part, applications are shielded from the implementation details of this processing pipeline. However, since each stage of the pipeline takes finite time and could potentially involve some buffering, applications need a method to check that all the commands they have issued have completed.

One way to achieve this is with the `glFinish` routine. This routine can be quite inefficient, so use it sparingly. It requires notification from the lowest levels of the rendering system that all the commands have been processed, all state changes have been fully realized, and that everything that should be displayed is displayed. It may be necessary to use `glFinish` in order to synchronize IVL rendering with rendering commands from Xlib or Motif.

Using the `glFlush` Routine

Sometimes it is only necessary to tell IVL to flush any/all internal command buffers so that all commands will be sent to the display hardware for processing. Use the `glFlush` routine to achieve this.

This routine can be more efficient than `glFinish`. It does not wait until the results of all the commands are completed, but just issues commands to flush any and all buffers in the system so that all pending commands are at least queued up for processing. The results of subsequent rendering commands are guaranteed to complete in a finite amount of time after the issuing of a `glFlush` call.

In a system with rendering hardware that is independent of the CPU, it may be possible for the CPU to continue processing while the rendering hardware completes processing the rendering commands. Therefore, applications should call `glFlush` rather than `glFinish` whenever possible.

Note: the `glFlush` routine can be more efficient than `glFinish`, but should still be used sparingly for highest application performance.

The following routines implicitly invoke `glFlush`:

- `glXSwapBuffers` causes `glFlush` to be called before it returns.
- Query routines such as `glGet` and `glIsEnabled` are only required to flush as much of the stream as is necessary in order to return valid results. These query routines do not guarantee that all pending rendering commands will be flushed.

5

Chapter 5: IVL Implementation and Device-Specific Information

List of Devices

The following device descriptions are included in this chapter:

- Entry-Level Color Graphics Devices:
  - Internal color graphics
  - HP VISUALIZE-EG (enhanced color) graphics
- The HCRX family of devices:
  - HCRX-8
  - HCRX-8Z
  - HP VISUALIZE-8
  - HCRX-24
  - HCRX-24Z
  - HP VISUALIZE-24
- The HCRX family of devices with IVX hardware:
  - HCRX-8 with IVX
  - HCRX-24 with IVX

See the *Graphics Administration Guide* for information about which workstation models support which graphics devices.

IVL Implementations

IVL functionality using the software implementation is identical to that of the hardware implementation.

The following graphics devices use the software implementation of IVL:

- Entry-Level Color Graphics Devices.
- The HCRX family of devices without IVX hardware.

The following graphics device uses the hardware implementation of IVL whenever possible:

- HCRX-8 devices with IVX hardware.
- HCRX-24 devices with IVX hardware.

Renderer Names

If you set the *name* argument of `glGetString` to `GL_RENDERER`, IVL returns the name of the renderer used on the current workstation. The currently supported return values are:

`Internal Color Graphics`
  Internal Color Graphics devices.
`Enhanced Color Graphics`
  HP VISUALIZE-EG devices.
`HCRXB8`
  HCRX-8 workstations.
`HCRXB8-Z`
  HCRX-8Z workstations.
`VISUALIZE-8`
  HP VISUALIZE-8 workstations.
`HCRXB24`
  HCRX-24 workstations.
`HCRXB24-Z`
  HCRX-24Z workstations.
`VISUALIZE-24`
  HP VISUALIZE-24 workstations.
`HCRXB8-IVX`
  HCRX-8 workstations with IVX hardware.
`HCRXB24-IVX`
  HCRX-24 workstations with IVX hardware.
`VISUALIZE-48`
  HP VISUALIZE-48 and HP VISUALIZE 48 XP workstations.
`Memory Driver`
  CRX-48Z; remote X drawable; or any unrecognized device.

Entry-Level Color Graphics Devices

The Internal Color Graphics device is identical in functionality to the HP VISUALIZE-EG device. The only difference you should see is faster performance on the HP VISUALIZE-EG device.

For the remainder of this chapter, information that describes Entry-Level Color Graphics devices applies to both the HP VISUALIZE-EG device and the Internal Color Graphics device.

Device Description

Entry-Level Color Graphics devices are found on Series 700 workstations that have graphics hardware capabilities on their SPU motherboard. In some cases, they are also available on plug-in cards.

The Entry-Level Color Graphics devices display color from a single bank of eight planes, supporting up to 256 colors at a time. They have two hardware color maps to reduce the likelihood of "technicolor" effects (which occur when two or more applications compete for entries in a single hardware color map).

See the *Graphics Administration Guide* for information on pixel resolution and refresh rates for these and other devices.

Supported Visuals

The following visuals are supported by IVL on the Entry-Level Color Graphics devices:

- PseudoColor (depth 8)
- GrayScale (depth 8)

Note that only one of the above visuals is supported at a time. The PseudoColor visual is used by default. If you boot your workstation with a grayscale monitor type, X11 will initialize itself to a grayscale mode. (This mode will exclude all access to color visuals. When initialized in a color mode, the X11 server will support only color visuals.)

Color Map Management

Many applications use the default X11 color map. A "technicolor" effect in the windows using the default color map may occur if a non-default color map is downloaded into the hardware color map that had previously contained the default color map.

Because many applications are likely to use the default X11 color map, and because the Entry-Level Color Graphics devices have two hardware color maps, the default behavior on these devices is to dedicate one hardware color map to always hold the default X11 color map. The second hardware color map is available to applications that use color maps other than the default.

This behavior can still cause the "technicolor" effect if two or more applications use different, non-default color maps. For example, application A uses the default X11 color map, application B uses a different color map, and application C uses a third color map. If applications A, B, and C all execute simultaneously on an Entry-Level Color Graphics device, application A would look correct. Either application B or C would show the technicolor effect; the application whose color map was last downloaded into the second hardware color map would look correct.

Overlay Transparency

Since there are no overlay planes, overlay transparency is not supported on the Entry-Level Color Graphics devices.

HCRX Family Device Descriptions

This section describes the HCRX family of devices, including the HP VISUALIZE devices. This information applies to HCRX devices with or without IVX hardware. The following devices are included in the HCRX family:

- HCRX-8
- HCRX-8Z
- HP VISUALIZE-8
- HCRX-24
- HCRX-24Z
- HP VISUALIZE-24

Note that only the HCRX-8 and HCRX-24 can support IVX hardware.

The graphics accelerators on these devices do not affect IVL performance or functionality. So, for the rest of this chapter, all references to HCRX-8 will apply to HCRX-8, HCRX-8Z, and HP VISUALIZE-8 devices, and all references to HCRX-24 apply to HCRX-24, HCRX-24Z, and HP VISUALIZE-24 devices.

These devices are all similar, and include hardware support for the following operations:

- Writing pixels to the frame buffer.
- Moving a block of pixels from one place in the frame buffer to another.
- Overlay plane transparency.
- Clipping.

Device Descriptions

HCRX-8 Description

The HCRX-8 device is a color display with two image-plane banks of 8 planes each. It supports 8 planes single-buffered or 8/8 planes double-buffered in the image planes. The image planes include two hardware color maps.

The HCRX-8 also has 8 overlay planes. It includes two hardware color maps in the overlay planes. The overlay color maps do not support transparency by default.

The default overlay visual has 256 entries per color map and no overlay transparency. When overlay transparency is enabled, the default overlay visual has 252 entries per color map. See the information below about color map limitations when using overlay transparency on the HCRX-8 devices.

See the *Graphics Administration Guide* for information on pixel resolution and refresh rates for these and other devices.

HCRX-24 Description

The HCRX-24 device is a color display with one image-plane bank of 24 planes. The image planes include two hardware color maps.

The HCRX-24 supports the following image-plane buffer modes:

- 8 planes single-buffered
- 8/8 double-buffered
- 12 planes single-buffered
- 12/12 double-buffered
- 24 planes single-buffered

The HCRX-24 also has 8 overlay planes. It includes two hardware color maps in the overlay planes. One of the overlay color maps supports transparency by default.

The default overlay visual has 256 entries per color map and no overlay transparency. The second overlay visual has 255 entries per color map and supports overlay transparency. See the information below about using overlay transparency on the HCRX-24 devices.

See the *Graphics Administration Guide* for information on pixel resolution and refresh rates for these and other devices.

Supported Visuals

HCRX-8. The following visuals are supported by IVL on the HCRX-8 in the overlay planes:

- PseudoColor (depth 8)

The following visuals are supported by IVL on the HCRX-8 in the image planes:

- PseudoColor (depth 8)

HCRX-24. The following visuals are supported by IVL on the HCRX-24 in the overlay planes:

- PseudoColor (depth 8)

The following visuals are supported by IVL on the HCRX-24 in the image planes:

- PseudoColor (depth 8)
- DirectColor (depth 12 or 24)
- TrueColor (depth 12 or 24)

Color Map Management

The information in this section applies to both the HCRX-8 and HCRX-24 devices.

Because so many applications use the default X11 color map, and because the HCRX devices have two hardware color maps in the overlay planes, the behavior on these devices is to dedicate (that is, lock) one overlay hardware color map to always hold the default X11 color map. This means that the assigned default overlay hardware color map cannot have another color map downloaded to it. The other overlay hardware color map is available to applications that use color maps other than the default.

Changing the Default Visual. By default, the default visual (where the root window and default color map reside) is in the overlay planes. Also by default, the overlay planes have the default X11 color map permanently locked into one hardware color map, and the second hardware color map is available for applications to use.

You can change this default mode by moving the default visual into the image planes. Doing this will limit the number of hardware color maps available to you. In this mode, HCRX devices provide a single hardware color map in the overlay planes.

To move the default visual into the image planes, edit your `X*screens` file, and add "`depth 8 doublebuffer`" to the line for your special device file. For example, if your `X0screens` file has the following line for its special device file:

```
/dev/crt
```

then you should change the line to read:

```
/dev/crt depth 8 doublebuffer
```

Overlay Transparency

Overlay Transparency with HCRX-8 Devices. Overlay transparency mode is not available on HCRX-8 devices by default. Enabling overlay transparency mode on an HCRX-8 will limit your system to one hardware color map in the overlay planes and one hardware color map in the image planes. This will increase the likelihood of seeing a "technicolor" effect.

To enable transparency, set the Screen Option `EnableOverlayTransparency`, then restart the X server.

With this mode enabled, color maps created in the default visual have 256 entries, with entry 255 reserved for transparency. Remember that if transparency is not enabled, only 252 entries are available in the color map.

Overlay Transparency with HCRX-24 Devices. Unlike the HCRX-8 devices, overlay transparency is available by default on the HCRX-24 devices, and using overlay transparency on the HCRX-24 devices does not change the number of available hardware color maps.

To create an overlay color map that supports transparency, create the color map using the visual that has transparency in its `SERVER_OVERLAY_VISUALS` property. The default overlay visual has a transparent type of `0` (`None`), and the transparent overlay visual has a transparent type of `1` (`TransparentPixel`). See the file `/usr/lib/X11/Xserver/info/screens/hp` for more information.

In overlay color maps that support transparency, the number of color map entries will change from 256 to 255 because the last entry becomes the transparent color map value. If your application requires that you have 256 entries in your color map, you need to set the `HP_COUNT_TRANSPARENT_IN_OVERLAY_VISUAL` environment variable to any value (for example, `TRUE`) before starting the X11 server. The X11 server will ignore any attempt to modify entry 255 of the color map.

Image Visualization Accelerator Device Description

This section describes behavior that is specific to HCRX devices with Image Visualization Accelerator (IVX) hardware. See the "HCRX Family Device Descriptions" section for general information on these devices. Note that IVX is supported on the HCRX-8 and HCRX-24 devices. IVX is not supported on the HCRX-8Z, VISUALIZE-8, HCRX-24Z, and VISUALIZE-24 devices.

Device Description

IVX accelerates many image processing operations in IVL. IVX is an accelerator that provides hardware support for the following:

- Input data formatting
- Convolution (with 3×3 kernel)
- Pan and zoom
- Rotations
- Bicubic interpolation
- Bilinear interpolation
- Nearest-neighbor interpolation
- **Window-level mapping**
- Rasterization
- Window clipping
- Scissor operations

Angle of Rotation

The angle of rotation is limited to increments of 0.1 degrees. You can supply any value, but it may be modified for rendering. For example, an angle of 23.89 degrees will be modified to 23.9 degrees for rendering. The angle of 23.89 degrees will be maintained by the API state, even though it renders at 23.9 degrees.

Rasterization

IVX rasterizes image data into the frame buffer from the bottom to the top of a window, and left to right across a scan line. This differs from other HP raster graphics devices, which rasterize from the top to the bottom of a window. The bottom-to-top order matches the semantics of the OpenGL API.

Performance Hints

Clip Rectangles

The performance of IVX will degrade as four or more clip rectangles obscure your image. This can be caused by software-specified clip boundaries or overlapping windows. If these conditions exist, multiple passes will be required to draw the same image.

Software versus Hardware-Accelerated Paths

As previously mentioned, HCRX devices with IVX hardware will use the accelerated hardware implementation of IVL whenever possible. The following conditions will force these devices to use the unaccelerated software path instead:

- If post-convolution bias is used.
- If the convolution kernel coefficient is 16.0 or greater.
- If the scale factor is outside the range of positive or negative 1.0 to 32.0, inclusive.
- For rotation angles that are not a multiple of 90 degrees, the absolute value of the ratio of the x zoom factor and the y zoom factor must be less than 2:1. In other words, the following condition should hold: $0.5 \leq |zoom_x|/|zoom_y| \leq 2.0$.
- The image size must be between 8×4 ($width \times height$) pixels and 32768×32768 pixels, inclusive.
- If your input format is `GL_RGBA` in a depth-24 visual, the hardware path will only be used if convolution is disabled and window-level mapping is not used.

Also, only drawing operations use the hardware path. Read and copy operations use the software path.

A

Appendix A: Quick Syntax Summary for IVL

IVL Rendering Routines

`glClear(`*mask*`)`

`glClearColor(`*red*, *green*, *blue*, *alpha*`)`

`glColorTableEXT(`*target*, *internalFormat*, *width*, *format*, *type*, **table*`)`

`glColorTableParameterEXTfv(`*target*, *pname*, **params*`)`

`glColorTableParameterEXTiv(`*target*, *pname*, **params*`)`

`glConvolutionFilter2DEXT(`*target*, *internalFormat*, *width*, *height*, *format*, *type*, **image*`)`

`glConvolutionParameterfEXT(`*target*, *pname*, *param*`)`

`glConvolutionParameterfvEXT(`*target*, *pname*, **params*`)`

`glConvolutionParameteriEXT(`*target*, *pname*, *param*`)`

`glConvolutionParameterivEXT(`*target*, *pname*, **params*`)`

`glCopyPixels(`*x*, *y*, *width*, *height*, *type*`)`

`glDisable(`*cap*`)`

`glDrawBuffer(`*mode*`)`

`glDrawPixels(`*width*, *height*, *format*, *type*, **pixels*`)`

`glEnable(`*cap*`)`

`glFinish(`*void*`)`

`glFlush(`*void*`)`

```
glGetBooleanv(pname, *params)

glGetColorTableEXT(target, format, type, *table)

glGetColorTableParameterfvEXT(target, pname, *params)

glGetColorTableParameterivEXT(target, pname, *params)

glGetConvolutionFilterEXT(target, format, type, *image)

glGetConvolutionParameterfvEXT(target, pname, *params)

glGetConvolutionParameterivEXT(target, pname, *params)

glGetDoublev(pname, *params)

GLenum glGetError(void)

glGetFloatv(pname, *params)

glGetImageTransformParameterfvHP(target, pname, *params)

glGetImageTransformParameterivHP(target, pname, *params)

glGetIntegerv(pname, *params)

const GLubyte *glGetString(name)

glImageTransformParameterfHP(target, pname, param)

glImageTransformParameterfvHP(target, pname, *params)

glImageTransformParameteriHP(target, pname, param)

glImageTransformParameterivHP(target, pname, *params)

GLboolean glIsEnabled(cap)

glPixelStoref(pname, param)

glPixelStorei(pname, param)

glPixelTransferf(pname, param)

glPixelTransferi(pname, param)
```

```
glRasterPos2i(x, y)

glRasterPos2iv(*v)

glReadBuffer(mode)

glReadPixels(x, y, width, height, format, type, *pixels)

glScissor(x, y, width, height)
```

GLX Utility Routines

```
XVisualInfo* glXChooseVisual(*dpy, screen, *attribList)

GLXContext glXCreateContext(*dpy, *vis, shareList, direct)

GLXPixmap glXCreateGLXPixmap(*dpy, *vis, pixmap)

glXDestroyContext(*dpy, ctx)

glXDestroyGLXPixmap(*dpy, pix)

int glXGetConfig(*dpy, *vis, attrib, *value)

Bool glXMakeCurrent(*dpy, drawable, ctx)

Bool glXQueryExtension(*dpy, *errorBase, *eventBase)

glXSwapBuffers(*dpy, drawable)
```

B

Appendix B: HP-IVL Reference

This portion of the document contains the reference pages for all the IVL routines.

glClear

Clear buffers to preset values.

C Specification

```
void glClear(GLbitfield mask)
```

Parameters

mask	Bitwise OR of masks that indicate the buffers to be cleared. The only supported value for *mask* is `GL_COLOR_BUFFER_BIT`.

Description

`glClear` sets the drawing area of the window to values previously selected by `glClearColor`.

The pixel ownership test and the scissor test affect the operation of `glClear`. The scissor box bounds the cleared region.

`glClear` takes a single argument that indicates which buffer to clear.

The only currently supported value of *mask* is:

- `GL_COLOR_BUFFER_BIT`
  Indicates the buffers currently enabled for color writing.

Notes

If a buffer is not present, then a `glClear` directed at that buffer has no effect.

Errors

`GL_INVALID_VALUE` is generated if any bit other than `GL_COLOR_BUFFER_BIT` is set in *mask*.

Associated Gets

glGet (GL_COLOR_CLEAR_VALUE)

See Also

glClearColor,
glDrawBuffer,
glGet,
glScissor.

glClearColor

Specify color values used for clearing the color buffers.

C Specification

```
void glClearColor(GLclampf red,
                  GLclampf green,
                  GLclampf blue,
                  GLclampf alpha)
```

Parameters

red, *green*, *blue*, *alpha*	Specify the red, green, blue, and alpha values used when the color buffers are cleared.

Description

`glClearColor` specifies the *red*, *green*, *blue*, and *alpha* values used by `glClear` to clear the color buffers. Values specified by `glClearColor` are clamped to the range [0,1].

Defaults

The default values for *red*, *green*, *blue*, and *alpha* are all zero.

Associated Gets

`glGet` (`GL_COLOR_CLEAR_VALUE`)

B

See Also

`glClear`,
`glGet`.

glColorTableEXT

Define a color lookup table.

C Specification

```
void glColorTableEXT(GLenum         target,
                     GLenum         internalFormat,
                     GLsizei        width,
                     GLenum         format,
                     GLenum         type,
                     const GLvoid *table)
```

Parameters

target — Must be GL_POST_IMAGE_TRANSFORM_COLOR_TABLE_HP.

internalFormat — The internal format of the color table. The allowable values are:

- GL_LUMINANCE,
- GL_LUMINANCE4_EXT,
- GL_LUMINANCE8_EXT,
- GL_LUMINANCE12_EXT,
- GL_LUMINANCE16_EXT,
- GL_RGBA,
- GL_RGBA2_EXT,
- GL_RGBA4_EXT,
- GL_RGB5_A1_EXT,
- GL_RGBA8_EXT,
- GL_RGB10_A2_EXT,
- GL_RGBA12_EXT, and
- GL_RGBA16_EXT.

width — The number of entries in the color lookup table specified by *table*.

format — The format of the pixel data in *table*. The allowable values are GL_LUMINANCE and GL_RGBA.

type	The type of the pixel data in *table*. The allowable values are GL_UNSIGNED_BYTE and GL_UNSIGNED_SHORT.
table	Pointer to the pixel data that will be processed to build the color table.

Description

glColorTableEXT is part of the EXT_color_table extension. At present, only the subset of EXT_color_table needed to support HP_image_transform has been implemented.

If target is GL_POST_IMAGE_TRANSFORM_COLOR_TABLE_HP, glColorTableEXT builds a color lookup table from an array of pixels. The pixel array specified by *width*, *format*, *type*, and *table* is extracted from memory and processed just as if glDrawPixels were called, but processing stops after the final expansion to RGBA is completed.

The R, G, B, and A components of each pixel are then scaled by the four GL_COLOR_TABLE_SCALE_EXT parameters and biased by the four GL_COLOR_TABLE_BIAS_EXT parameters. (Use glColorTableParameterEXT to set the scale and bias parameters.) The R, G, B, and A values are then clamped to the range [0,1].

Each pixel is then converted to the internal format specified by *internalFormat*. This conversion simply maps the component values of the pixel (R, G, B, and A) to the values included in the internal format (red, green, blue, alpha, luminance, and intensity). If *internalFormat* is GL_RGBA, then the R, G, B, and A components are mapped to the R, G, B, and A components of the internal format. If *internalFormat* is GL_LUMINANCE, then the R component is mapped to the luminance component of the internal format.

The luminance and RBGA variants are handled in the same way as their base value. For example, GL_LUMINANCE4_EXT is handled in the same way as GL_LUMINANCE, and GL_RGBA8_EXT is handled in the same way as GL_RGBA. It is permissible for implementations to allocate storage in a fashion other than what was specifically requested by *internalFormat*. Therefore the value of *internalFormat* is more a hint than an exact allocation specification.

Finally, the red, green, blue, alpha, and/or luminance components of the resulting pixels are stored in the color table. They form a one-dimensional table with indices in the range [0, *width*−1].

Notes

For GL_POST_IMAGE_TRANSFORM_COLOR_TABLE_HP, *width* must be a power of two.

The combination of *format* GL_RGBA and *type* GL_UNSIGNED_SHORT is not supported in this release.

Errors

GL_INVALID_ENUM is generated if *target* is not one of the allowable values.

GL_INVALID_ENUM is generated if *internalFormat* is not one of the allowable values.

GL_INVALID_VALUE is generated if *width* is less than zero or is not a power of 2.

GL_INVALID_ENUM is generated if *format* is not one of the allowable values.

GL_INVALID_ENUM is generated if *type* is not one of the allowable values.

GL_TABLE_TOO_LARGE_EXT is generated if the requested color table is too large to be supported by the implementation.

Associated Gets

glGetColorTableParameterEXT

See Also

glColorTableParameterEXT,
glGetColorTableParameterEXT.

glColorTableParameter*vEXT

Set color lookup table parameters.

C Specification

```
void glColorTableParameterfvEXT(GLenum          target,
                                GLenum          pname,
                                const GLfloat *params)

void glColorTableParameterivEXT(GLenum        target,
                                GLenum        pname,
                                const GLint *params)
```

Parameters

target	The target color table. Must be `GL_POST_IMAGE_TRANSFORM_COLOR_TABLE_HP`.
pname	The symbolic name of a color lookup table parameter. Must be either `GL_COLOR_TABLE_SCALE_EXT` or `GL_COLOR_TABLE_BIAS_EXT`.
params	A pointer to an array where the values of the parameters are stored.

Description

`glColorTableParameterEXT` is part of the `EXT_color_table` extension, which adds several color lookup tables to the pixel transfer path. At present, only the subset of `EXT_color_table` needed to support the `HP_image_transform` extension has been implemented.

`glColorTableParameterEXT` specifies the scale factors and bias terms applied to color components when they are loaded into the color table. The *target* argument must be `GL_POST_IMAGE_TRANSFORM_COLOR_TABLE_HP`.

The *pname* argument must be `GL_COLOR_TABLE_SCALE_EXT` to set the scale factors. In this case, *params* points to an array of four values, which are the scale factors for red, green, blue, and alpha, in that order.

The *pname* argument must be GL_COLOR_TABLE_BIAS_EXT to set the bias terms. The *params* argument points to an array of four values, which are the bias terms for red, green, blue, and alpha, in that order.

Calling glColorTableEXT whenever a table is loaded will result in applying the scale and bias values to the color lookup table values.

The post-image transform color lookup table is specified by glColorTableEXT, using GL_POST_IMAGE_TRANSFORM_COLOR_TABLE_HP as the target.

Errors

GL_INVALID_ENUM is generated if *target* or *pname* is not an acceptable value.

Associated Gets

glGetColorTableParameterEXT

See Also

glColorTableEXT,
glGet,
glGetColorTableParameterEXT.

glConvolutionFilter2DEXT

Define a two-dimensional convolution filter.

C Specification

```
void glConvolutionFilter2DEXT(GLenum        target,
                              GLenum        internalFormat,
                              GLsizei       width,
                              GLsizei       height,
                              GLenum        format,
                              GLenum        type,
                              const GLvoid *image)
```

Parameters

target	Must be `GL_CONVOLUTION_2D_EXT`.
internalFormat	The internal format of the convolution filter kernel. The allowable values are `GL_LUMINANCE` and `GL_RGBA`.
width	The width of the pixel array referenced by *image*.
height	The height of the pixel array referenced by *image*.
format	The format of the pixel data in *image*. The allowable values are `GL_LUMINANCE` and `GL_RGBA`.
type	The type of the pixel data in *image*. The only allowable value is `GL_FLOAT`.
image	Pointer to a two-dimensional array of pixel data that is processed to build the convolution filter kernel.

Description

`glConvolutionFilter2DEXT` builds a two-dimensional convolution filter kernel from an array of pixels.

The pixel array specified by *width*, *height*, *format*, *type*, and *image* is extracted from memory and processed just as if `glDrawPixels` were called, but processing stops after completing the final expansion to RGBA.

The R, G, B, and A components of each pixel are next scaled by the four 2D GL_CONVOLUTION_FILTER_SCALE_EXT parameters and biased by the four 2D GL_CONVOLUTION_FILTER_BIAS_EXT parameters. (The scale and bias parameters are set by glConvolutionParameterEXT using the GL_CONVOLUTION_2D_EXT *target* and the *names* GL_CONVOLUTION_FILTER_SCALE_EXT and GL_CONVOLUTION_FILTER_BIAS_EXT. The parameters themselves are vectors of four values that are applied to red, green, blue, and alpha, in that order.) The R, G, B, and A values are not clamped to [0,1] at any time during this process.

Each pixel is then converted to the internal format specified by *internalFormat*. This conversion simply maps the component values of the pixel (R, G, B, and A) to the values included in the internal format (red, green, blue, alpha, luminance, and intensity). If *internalFormat* is GL_RGBA, then the R, G, B, and A components are mapped to the R, G, B, and A components of the internal format. If *internalFormat* is GL_LUMINANCE, then the R component maps to the luminance component of the internal format.

The red, green, blue, alpha, and/or luminance components of the resulting pixels are stored in floating-point rather than integer format. They form a two-dimensional filter kernel image indexed with coordinates i and j such that i starts at zero and increases from left to right, and j starts at zero and increases from bottom to top. Kernel location i,j is derived from the Nth pixel, where N is $i+j\,width$.

Note that after performing a convolution, the resulting color components are also scaled by their corresponding GL_POST_CONVOLUTION_c_SCALE_EXT parameters and biased by their corresponding GL_POST_CONVOLUTION_c_BIAS_EXT parameters (where c takes on the values RED, GREEN, BLUE, and ALPHA). These parameters are set by glPixelTransfer.

Errors

GL_INVALID_ENUM is generated if *target* is not GL_CONVOLUTION_2D_EXT.

GL_INVALID_ENUM is generated if *internalFormat* is not one of the allowable values.

GL_INVALID_VALUE is generated if *width* is less than zero or greater than the maximum supported value. This value may be queried with

glGetConvolutionParameterEXT using *target* GL_CONVOLUTION_2D_EXT and name GL_MAX_CONVOLUTION_WIDTH_EXT.

GL_INVALID_VALUE is generated if *height* is less than zero or greater than the maximum supported value. This value may be queried with

glGetConvolutionParameterEXT using *target* GL_CONVOLUTION_2D_EXT and name GL_MAX_CONVOLUTION_HEIGHT_EXT.

GL_INVALID_ENUM is generated if *format* is not one of the allowable values.

GL_INVALID_ENUM is generated if *type* is not one of the allowable values.

Associated Gets

glGetConvolutionFilterEXT
glGetConvolutionParameterEXT.

Notes

The *width* and *height* arguments must both be set to 3 for this release. Values of 0, 1, or 2 will cause a GL_INVALID_VALUE error to be generated.

See Also

glConvolutionParameterEXT,
glEnable (with parameter GL_CONVOLUTION_2D_EXT),
glDrawPixels,
glGetConvolutionFilterEXT,
glGetConvolutionParameterEXT,
glPixelTransfer.

glConvolutionParameter*EXT

Set convolution parameters.

C Specification (for Single-Value Attributes)

```
void glConvolutionParameterfEXT(GLenum  target,
                                GLenum  pname,
                                GLfloat param)

void glConvolutionParameteriEXT(GLenum target,
                                GLenum pname,
                                GLint  param)
```

Parameters

target The target for the convolution parameter. Must be `GL_CONVOLUTION_2D_EXT`.

pname The parameter to be set. Must be `GL_CONVOLUTION_BORDER_MODE_EXT`.

param The parameter value. Must be one of:

- `GL_REDUCE_EXT`,
- `GL_IGNORE_BORDER_HP`,
- `GL_CONSTANT_BORDER_HP`,
- `GL_WRAP_BORDER_HP`, or
- `GL_REPLICATE_BORDER_HP`.

C Specification (for Multiple-Value Attributes)

```
void glConvolutionParameterfvEXT(GLenum         target,
                                 GLenum         pname,
                                 const GLfloat *params)

void glConvolutionParameterivEXT(GLenum       target,
                                 GLenum       pname,
                                 const GLint *params)
```

Parameters

target The target for the convolution parameter. Must be GL_CONVOLUTION_2D_EXT.

pname The parameter to be set. Must be one of

- GL_CONVOLUTION_FILTER_SCALE_EXT,
- GL_CONVOLUTION_FILTER_BIAS_EXT,
- GL_CONVOLUTION_BORDER_MODE_EXT,
- GL_CONVOLUTION_BORDER_COLOR_HP.

params The parameter value. If *pname* is GL_CONVOLUTION_BORDER_MODE_EXT, *params* must be one of:

GL_REDUCE_EXT,
GL_IGNORE_BORDER_HP,
GL_CONSTANT_BORDER_HP,
GL_WRAP_BORDER_HP, or
GL_REPLICATE_BORDER_HP.

Otherwise, *params* must be a vector of four values (for red, green, blue, and alpha, respectively). This vector specifies values for scaling (when *pname* is GL_CONVOLUTION_FILTER_SCALE_EXT) or for biasing (when *pname* is GL_CONVOLUTION_FILTER_BIAS_EXT) a convolution filter kernel. Or, the *params* value will be used as the border color (when *pname* is GL_CONVOLUTION_BORDER_COLOR_HP).

Description

glConvolutionParameterEXT sets the value of a convolution parameter.

The *target* argument selects the convolution filter to be affected, and must be GL_CONVOLUTION_2D_EXT for the 2D filter.

The *pname* argument selects the parameter to be changed. Legal values for *pname* are:

- GL_CONVOLUTION_FILTER_SCALE_EXT
  If used, values pointed at by *params* become the current convolution filter scaling values for the specified target. The convolution scale values are

applied to convolution filter values whenever the convolution filter is set using glConvolutionFilter2DEXT.

- GL_CONVOLUTION_FILTER_BIAS_EXT
  If used, values pointed at by *params* become the current convolution filter bias values for the specified target. The convolution filter bias values are applied to convolution filter values whenever the convolution filter is set using glConvolutionFilter2DEXT.

- GL_CONVOLUTION_BORDER_COLOR_HP
  If used, the value in *params* becomes the current convolution border color for the specified target. The convolution border color is used along the edges of a convolved image when the convolution border mode is set to GL_CONSTANT_BORDER_HP.

- GL_CONVOLUTION_BORDER_MODE_EXT
  If used, the value in *params* becomes the current convolution border mode for the specified target. For the purpose of the following discussion, the width and height of the current convolution filter are specified by Wf and Hf, and the width and height of the source image are specified by Ws and Hs. The symbols Cw and Ch indicate half the width and height of the current convolution filter and are defined as $Cw = \left\lfloor \frac{Wf}{2} \right\rfloor$ and $Ch = \left\lfloor \frac{Hf}{2} \right\rfloor$. The legal values for *params* are:

  - GL_REDUCE_EXT
    If used, edges are eliminated, so the convolved image becomes smaller than the input image. When this mode is in effect, the image resulting from convolution is smaller than the source image. The convolved image width will be $Ws - Wf + 1$ and height will be $Hs - Hf + 1$. (If this reduction would generate an image with zero or negative width and/or height, then there would be no image data left to process at this point in the pipeline. So, the output is simply null, with no error generated.) The image resulting from convolution has coordinates that range from zero through $Ws - Wf$ in width and zero through $Hs - Hf$ in height.

  - GL_IGNORE_BORDER_HP
    If the convolution border mode is GL_IGNORE_BORDER_HP, the output image has the same dimensions as the source image, but the resulting pixels in the Cw columns along the left and right edges will be the same as the corresponding pixels in the source image, and the pixels in the Ch rows

on the top and bottom edges will be the same as the corresponding pixels in the source image.

- GL_CONSTANT_BORDER_HP
  If the convolution border mode is GL_CONSTANT_BORDER_HP, the output image has the same dimensions as the source image, but the current convolution border color will be used as input for the convolution operation wherever no source image pixels exist. (This occurs in the *Cw* columns along the left and right edges and the *Ch* rows on the top and bottom edges.) The current convolution border color is set by calling glConvolutionParameterivEXT or glConvolutionParameterfvEXT with *pname* set to GL_CONVOLUTION_BORDER_COLOR_HP and *params* containing four values that comprise the RGBA color to be used as the image border. Integer color components are interpreted linearly such that the most positive integer maps to 1.0, and the most negative integer maps to −1.0. Floating point color components are clamped to the range [0,1] when they are specified.

- GL_WRAP_BORDER_HP
  If the convolution border mode is GL_WRAP_BORDER_HP, the output image has the same dimensions as the source image, and the source image is assumed to be continuously wrapped in both x and y directions. Therefore, source image pixels in the *Cw* columns on the right edge are used in the convolution computation for the *Cw* columns on the left edge of the image, and vice versa. Similarly, source image pixels in the *Ch* rows on the top of the image are used in the convolution computation for the *Ch* rows on the bottom of the image and vice versa.

- GL_REPLICATE_BORDER_HP
  If the convolution border mode is GL_REPLICATE_BORDER_HP, the output image has the same dimensions as the source image, and the source image is assumed to have replicated pixels along the borders. When computing the convolution along the border of the source image, the replicated border pixels are used wherever no source image pixels exist.

Defaults

For each potential target, the default convolution filter bias values are:

pname	Default Values
GL_CONVOLUTION_FILTER_SCALE_EXT	(1, 1, 1, 1)
GL_CONVOLUTION_FILTER_BIAS_EXT	(0, 0, 0, 0)
GL_CONVOLUTION_BORDER_COLOR_HP	(0, 0, 0, 0)

The default convolution border mode is GL_REDUCE_EXT.

Errors

GL_INVALID_ENUM is generated if *target* is not one of the allowable values.

GL_INVALID_ENUM is generated if *pname* is not one of the allowable values.

GL_INVALID_ENUM is generated if *pname* is
GL_CONVOLUTION_BORDER_MODE_EXT and *params* is not one of:

GL_REDUCE_EXT,
GL_IGNORE_BORDER_HP,
GL_CONSTANT_BORDER_HP,
GL_WRAP_BORDER_HP, or
GL_REPLICATE_BORDER_HP.

Associated Gets

glGetConvolutionParameterEXT

See Also

glConvolutionFilter2DEXT,
glGetConvolutionParameterEXT.

glCopyPixels

Copy pixels in the frame buffer.

C Specification

```
void glCopyPixels(GLint   x,
                  GLint   y,
                  GLsizei width,
                  GLsizei height,
                  GLenum  type)
```

Parameters

x, *y*	Specify the window coordinates of the lower left corner of the rectangular region of pixels to be copied.
width, *height*	Specify the dimensions of the rectangular region of pixels to be copied. Both must be non-negative.
type	Specifies the type of source buffer (color, depth, or stencil). Only the symbolic constant `GL_COLOR` is allowed.

Description

`glCopyPixels` copies a screen-aligned rectangle of pixels from the specified frame buffer location to a region relative to the current raster position. Its operation is well defined only if the entire pixel source region is within the exposed portion of the window. Results of copies from outside the window, or from regions of the window that are not exposed, are hardware-dependent and undefined.

B

The *x* and *y* arguments specify the window coordinates of the lower left corner of the rectangular region to be copied. The *width* and *height* arguments specify the dimensions of the rectangular region to be copied. Both *width* and *height* must be non-negative.

`glCopyPixels` copies values from each pixel with the lower left-hand corner at $(x+i, y+j)$ for $0<i\leq width$ and $0\leq j<height$. This pixel is said to be the ith pixel in the jth row. Pixels are copied in row order from the lowest to the highest row, left to right across each row.

The *type* argument specifies the type of data to be copied. The only currently allowable value for *type* is GL_COLOR. Details for this data type are as follows:

GL_COLOR RGBA colors are read from the buffer currently specified as the read source buffer (see glReadBuffer).

The red, green, blue, and alpha components of each pixel that is read are converted to an internal floating-point format with unspecified precision. The conversion maps the largest representable component value to 1.0, and component value zero to 0.0.

If convolution is enabled by calling glEnable with GL_CONVOLUTION_2D_EXT, the rectangle of pixel values being copied will be convolved with the current 2D convolution filter kernel. The behavior of the convolution operation is controlled by the convolution parameters set by glConvolutionParameterEXT. As part of the convolution operation, IVL will also apply the post-convolution scale and bias values (set by calling glPixelTransfer).

Next, if image transformation is enabled, IVL will scale, rotate, and translate the pixel rectangle being copied according to the image transformation parameters set by glImageTransformParameterHP.

Following this, if the post-image transform color table is enabled, pixel values will undergo a table lookup operation. The values in this lookup table are established by calling glColorTableEXT.

The resulting RGBA colors are then converted to fragments by assigning window coordinates $(xr+i, yr+j)$, where (xr, yr) is the current raster position, and the pixel was the ith pixel in the jth row. These pixel fragments are subsequently written to the frame buffer. As pixels are written, they are subjected to the pixel ownership test and, if enabled, the scissor test.

Examples

If all of the pixel transfer operations are disabled, the following command will copy the color pixel in the lower left corner of the window to the current raster position:

```
glCopyPixels(0, 0, 1, 1, GL_COLOR);
```

Notes

Modes specified by `glPixelStore` have no effect on the operation of `glCopyPixels`.

Errors

`GL_INVALID_ENUM` is generated if *type* is not an accepted value.

`GL_INVALID_VALUE` is generated if either *width* or *height* is negative.

Associated Gets

`glGet(GL_CURRENT_RASTER_POSITION)`
`glGet(GL_CURRENT_RASTER_POSITION_VALID)`

See Also

`glColorTableEXT`,
`glConvolutionParameterEXT`,
`glDrawBuffer`,
`glDrawPixels`,
`glEnable`,
`glGet`,
`glImageTransformParameterHP`,
`glPixelTransfer`,
`glRasterPos`,
`glReadBuffer`,
`glReadPixels`.

glDrawBuffer

Specify which color buffers are to be drawn into.

C Specification

```
void glDrawBuffer(GLenum mode)
```

Parameters

mode	Specifies which buffers are to be drawn into. Symbolic constants `GL_FRONT_LEFT`, `GL_BACK_LEFT`, `GL_FRONT`, and `GL_BACK` are accepted.

Description

When colors are written to the frame buffer, they are written into the color buffers specified by `glDrawBuffer`. The specifications are as follows:

`GL_FRONT_LEFT`	Only the front left color buffer is written.
`GL_BACK_LEFT`	Only the back left color buffer is written.
`GL_FRONT`	Only the front left and front right color buffers are written. If there is no front right color buffer, only the front left color buffer is written.
`GL_BACK`	Only the back left and back right color buffers are written. If there is no back right color buffer, only the back left color buffer is written.

Monoscopic contexts include only left buffers, and stereoscopic contexts include both left and right buffers. Likewise, single-buffered contexts include only front buffers, and double-buffered contexts include both front and back buffers. The context is selected at IVL initialization time.

Defaults

The default value of *mode* is GL_FRONT for single-buffered contexts, and GL_BACK for double-buffered contexts.

Notes

Stereo is not supported in the first release of IVL. Hence, GL_FRONT_LEFT is equivalent to GL_FRONT, and GL_BACK_LEFT is equivalent to GL_BACK. Use the buffer names that allow your application to work properly should it ever be run on a system that supports stereo windows. Non-stereo applications will typically use GL_FRONT and GL_BACK.

Errors

GL_INVALID_ENUM is generated if *mode* is not an accepted value.

GL_INVALID_OPERATION is generated if none of the buffers indicated by *mode* exists.

Associated Gets

glGet (GL_DRAW_BUFFER)

See Also

glGet,
glReadBuffer.

glDrawPixels

Write a block of pixels to the frame buffer.

C Specification

```
void glDrawPixels(GLsizei        width,
                  GLsizei        height,
                  GLenum         format,
                  GLenum         type,
                  const GLvoid *pixels)
```

Parameters

width, *height*	Specify the dimensions of the pixel rectangle that will be written into the frame buffer.
format	Specifies the format of the pixel data. Symbolic constants `GL_RGBA` and `GL_LUMINANCE` are accepted.
type	Specifies the data type for *pixels*. Symbolic constants `GL_UNSIGNED_BYTE` and `GL_UNSIGNED_SHORT` are accepted.
pixels	Specifies a pointer to the pixel data.

Description

`glDrawPixels` reads pixel data from memory and writes it into the frame buffer relative to the current raster position. Use `glRasterPos` to set the current raster position, and use `glGet` with argument `GL_CURRENT_RASTER_POSITION` to query the raster position.

Data is read from *pixels* as a sequence of unsigned bytes or unsigned shorts, depending on *type*. Each of these bytes or shorts is interpreted as one color component. Color components are treated as groups of one or four values, based on *format*. Groups of components are referred to as *pixels*.

The specified *widthheight* rectangle of pixels are read from memory, starting at location *pixels*. By default, these pixels are taken from adjacent memory locations, except that after all *width* pixels are read, the read pointer is advanced

to the next four-byte boundary. The four-byte row alignment is specified by `glPixelStore` with argument `GL_UNPACK_ALIGNMENT`, and it can be set to one, two, four, or eight bytes. Other pixel store parameters specify the number of bytes the read pointer should be advanced prior to reading the first row of pixels, and the number of bytes to advance the read pointer after reading each row. Refer to the `glPixelStore` reference page for details on these options.

The *widthheight* pixels that are read from memory are each transformed in the same way, based on the values of several parameters that affect pixel transfer operations. The details of these operations, as well as the target buffer into which the pixels are drawn, are specific to the format of the pixels, as specified by *format*. The *format* argument can assume one of two symbolic values:

`GL_RGBA` Each pixel is a four-component group: red first, followed by green, followed by blue, followed by alpha. Signed integer values are mapped linearly to an internal floating-point format with unspecified precision such that the most positive representable integer value maps to 1.0, and the most negative representable value maps to -1.0.

If convolution is enabled by calling `glEnable` with `GL_CONVOLUTION_2D_EXT`, the rectangle of pixel values being copied will be processed with the current 2D convolution filter kernel. The behavior of the convolution operation is controlled by the parameters set by `glConvolutionParameterEXT`. As part of the convolution operation, IVL will also apply the post-convolution scale and bias values (set by calling `glPixelTransfer`).

Next, if image transformation is enabled, IVL will also scale, rotate, and translate the pixel rectangle being copied according to the image transformation parameters set by `glImageTransformParameterHP`.

Following this, if the post-image-transform color table is enabled, pixel values will be undergo a table lookup operation. The values in this lookup table are established by calling `glColorTableEXT`.

The resulting RGBA colors are then converted to fragments by assigning window coordinates $(xr+i, yr+j)$, where (xr, yr)

is the current raster position, and the pixel was the ith pixel in the jth row. These pixel fragments are then written to the frame buffer. As pixels are written, they are subjected to the pixel ownership test and, if enabled, the scissor test.

`GL_LUMINANCE` Each pixel is a single luminance component. This component is converted to the internal floating-point format in the same way as the red component of an RGBA pixel is, then it is converted to an RGBA pixel with red, green, and blue set to the converted luminance value, and alpha set to 1.0. After this conversion, the pixel is treated just as if it had been read as an RGBA pixel.

The following table summarizes the meaning of the valid constants for the *type* parameter:

type	corresponding type
`GL_UNSIGNED_BYTE`	unsigned 8-bit integer
`GL_UNSIGNED_SHORT`	unsigned 16-bit integer

Notes

For this release, the combination of *format* `GL_RGBA` and *type* `GL_UNSIGNED_SHORT` is not supported.

Errors

`GL_INVALID_VALUE` is generated if either *width* or *height* is negative.

`GL_INVALID_ENUM` is generated if *format* or *type* is not one of the accepted values.

Associated Gets

`glGet` (`GL_CURRENT_RASTER_POSITION`)
`glGet` (`GL_CURRENT_RASTER_POSITION_VALID`)

glDrawPixels

See Also

glColorTableEXT,
glConvolutionParameterEXT,
glCopyPixels,
glEnable,
glGet,
glImageTransformParameterHP,
glPixelStore,
glPixelTransfer,
glRasterPos,
glReadPixels,
glScissor.

glEnable, glDisable

Enable or disable IVL capabilities.

C Specification

```
void glEnable(GLenum cap)

void glDisable(GLenum cap)
```

Parameters

cap	Specifies a symbolic constant indicating an IVL capability.

Description

`glEnable` and `glDisable` enable and disable various capabilities. Use `glIsEnabled` or `glGet` to determine the current setting of any capability.

Both `glEnable` and `glDisable` take a single argument, *cap*, which can assume one of the following values:

- `GL_CONVOLUTION_2D_EXT`
  If enabled, perform two-dimensional convolution during pixel transfers. See `glConvolutionFilter2DEXT`.
- `GL_SCISSOR_TEST`
  If enabled, discard fragments that are outside the scissor rectangle. See `glScissor`.
- `GL_POST_IMAGE_TRANSFORM_COLOR_TABLE_HP`
  If enabled, perform a color table lookup operation after the image transformation operation.
- `GL_IMAGE_TRANSFORM_2D_HP`
  If enabled, perform an image transformation operation as part of pixel transfer. See `glImageTransformParameterHP`.

glEnable, glDisable

Errors

GL_INVALID_ENUM is generated if *cap* is not one of the values listed above.

See Also

glColorTableEXT,
glConvolutionFilter2DEXT,
glImageTransformParameterHP,
glIsEnabled,
glScissor.

glFinish

Block until all IVL execution is complete.

C Specification

```
void glFinish(void void)
```

Description

`glFinish` does not return until the effects of all previously called IVL commands are complete. Such effects include all changes to IVL state, all changes to connection state, and all changes to the frame buffer contents.

Notes

`glFinish` may take more time than desired, since it must block until all rendering is completed. Whenever possible, use `glFlush` instead.

See Also

`glFlush`.

glFlush

Force execution of IVL commands in finite time.

C Specification

```
void glFlush(void void)
```

Description

Implementations of IVL on different platforms may buffer commands in several different locations, including network buffers and the graphics accelerator itself. `glFlush` empties all of these buffers, causing all issued commands to be executed as quickly as they are accepted by the actual rendering engine. Though this execution may not be completed in any particular time period, it does complete in finite time.

Because any IVL program might be executed over a network, or on an accelerator that buffers commands, all programs should call `glFlush` whenever they count on having all of their previously issued commands completed. For example, if user input depends on a generated image, a call to `glFlush` should be placed between the calls to `glDrawPixels` and `glReadPixels`.

Notes

`glFlush` can return at any time. It does not wait until the execution of all previously issued IVL commands is complete.

See Also

`glFinish`.

glGet*v

Return the value or values of a selected parameter.

C Specification

```
void glGetBooleanv(GLenum     pname,
                   GLboolean *params)

void glGetDoublev(GLenum    pname,
                  GLdouble *params)

void glGetFloatv(GLenum   pname,
                 GLfloat *params)

void glGetIntegerv(GLenum pname,
                   GLint *params)
```

Parameters

pname	Specifies the parameter value to be returned. The symbolic constants in the list below are accepted.
params	Returns the value or values of the specified parameter.

Description

These four commands return values for simple state variables in IVL. The *pname* argument is a symbolic constant indicating the state variable to be returned, and *params* is a pointer to an array of the indicated type in which to place the returned data.

Type conversion is performed if *params* has a different type than the state variable value being requested. If `glGetBooleanv` is called, a floating-point or integer value is converted to `GL_FALSE` if and only if it is zero. Otherwise, it is converted to `GL_TRUE`. If `glGetIntegerv` is called, Boolean values are returned as `GL_TRUE` or `GL_FALSE`, and most floating-point values are rounded to the nearest integer value. Floating-point colors, however, are returned with a linear mapping that

maps 1.0 to the most positive representable integer value, and −1.0 to the most negative representable integer value. If `glGetFloatv` or `glGetDoublev` is called, Boolean values are returned as `GL_TRUE` or `GL_FALSE`, and integer values are converted to floating-point values.

The following symbolic constants are accepted by *pname*:

- `GL_ALPHA_BITS`
  params returns one value, the number of alpha bitplanes in each color buffer.
- `GL_BLUE_BITS`
  params returns one value, the number of blue bitplanes in each color buffer.
- `GL_COLOR_CLEAR_VALUE`
  params returns four values: the red, green, blue, and alpha values used to clear the color buffers. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and −1.0 returns the most negative representable integer value. See `glClearColor`.
- `GL_CONVOLUTION_2D_EXT`
  params returns a single Boolean value indicating whether two-dimensional convolution will be performed during pixel transfers. See `glConvolutionFilter2DEXT`.
- `GL_CURRENT_RASTER_POSITION`
  params returns four values: the x, y, z, and w components of the current raster position. x, y, and z are in window coordinates, and w is in clip coordinates. See `glRasterPos`.
- `GL_CURRENT_RASTER_POSITION_VALID`
  params returns a single Boolean value indicating whether the current raster position is valid. See `glRasterPos`.

- `GL_DOUBLEBUFFER`
  params returns a single Boolean value indicating whether double-buffering is supported.
- `GL_DRAW_BUFFER`
  params returns one value, a symbolic constant indicating which buffers are being drawn to. See `glDrawBuffer`.

- GL_GREEN_BITS
  params returns one value, the number of green bitplanes in each color buffer.
- GL_PACK_ALIGNMENT
  params returns one value, the byte alignment used for writing pixel data to memory. See glPixelStore.
- GL_PACK_ROW_LENGTH
  params returns one value, the row length used for writing pixel data to memory. See glPixelStore.
- GL_PACK_SKIP_PIXELS
  params returns one value, the number of pixel locations skipped before the first pixel is written into memory. See glPixelStore.
- GL_PACK_SKIP_ROWS
  params returns one value, the number of rows of pixel locations skipped before the first pixel is written into memory. See glPixelStore.
- GL_POST_CONVOLUTION_ALPHA_BIAS_EXT
  params returns a single value, the bias term to be added to alpha immediately after convolution. See glConvolutionFilter2DEXT.
- GL_POST_CONVOLUTION_ALPHA_SCALE_EXT
  params returns a single value, the scale factor to be applied to alpha immediately after post-convolution scaling. See glConvolutionFilter2DEXT.
- GL_POST_CONVOLUTION_BLUE_BIAS_EXT
  params returns a single value, the bias term to be added to blue immediately after convolution. See glConvolutionFilter2DEXT.
- GL_POST_CONVOLUTION_BLUE_SCALE_EXT
  params returns a single value, the scale factor to be applied to blue immediately after post-convolution scaling. See glConvolutionFilter2DEXT.
- GL_POST_CONVOLUTION_GREEN_BIAS_EXT
  params returns a single value, the bias term to be added to green immediately after convolution. See glConvolutionFilter2DEXT.
- GL_POST_CONVOLUTION_GREEN_SCALE_EXT
  params returns a single value, the scale factor to be applied to green immediately after post-convolution scaling. See glConvolutionFilter2DEXT.

- GL_POST_CONVOLUTION_RED_BIAS_EXT
  params returns a single value, the bias term to be added to red immediately after convolution. See glConvolutionFilter2DEXT.
- GL_POST_CONVOLUTION_RED_SCALE_EXT
  params returns a single value, the scale factor to be applied to red immediately after post-convolution scaling. See glConvolutionFilter2DEXT.
- GL_READ_BUFFER
  params returns one value, a symbolic constant indicating which color buffer is selected for reading. See glReadPixels.
- GL_RED_BITS
  params returns one value, the number of red bitplanes in each color buffer.
- GL_RGBA_MODE
  params returns a single Boolean value indicating whether IVL is in RGBA mode (true) or color index mode (false).
- GL_SCISSOR_BOX
  params returns four values: the x and y window coordinates of the scissor box, followed by its width and height. See glScissor.
- GL_SCISSOR_TEST
  params returns a single Boolean value indicating whether scissoring is enabled. See glScissor.
- GL_UNPACK_ALIGNMENT
  params returns one value, the byte alignment used for reading pixel data from memory. See glPixelStore.
- GL_UNPACK_ROW_LENGTH
  params returns one value, the row length used for reading pixel data from memory. See glPixelStore.
- GL_UNPACK_SKIP_PIXELS
  params returns one value, the number of pixel locations skipped before the first pixel is read from memory. See glPixelStore.
- GL_UNPACK_SKIP_ROWS
  params returns one value, the number of rows of pixel locations skipped before the first pixel is read from memory. See glPixelStore.

- GL_POST_IMAGE_TRANSFORM_COLOR_TABLE_HP
  params returns a single value indicating whether the post-image transformation color lookup table is enabled. See glColorTableEXT.
- GL_IMAGE_TRANSFORM_2D_HP
  params returns a single value indicating whether the 2D image transformation operation is enabled. See glImageTransformParameterHP.

Many of the Boolean parameters can also be queried more easily using glIsEnabled.

Errors

GL_INVALID_ENUM is generated if *pname* is not an accepted value.

See Also

glClearColor,
glColorTableEXT,
glColorTableEXT,
glConvolutionFilter2DEXT,
glDrawBuffer,
glGetColorTableParameterEXT,
glGetConvolutionFilterEXT,
glGetConvolutionParameterEXT,
glGetError,
glGetImageTransformParameterHP,
glGetString,
glImageTransformParameterHP,
glIsEnabled,
glPixelStore,
glRasterPos,
glReadPixels,
glScissor.

glGetColorTableEXT

Retrieve the contents of a color lookup table.

C Specification

```
void glGetColorTableEXT(GLenum  target,
                        GLenum  format,
                        GLenum  type,
                        GLvoid *table)
```

Parameters

target	Must be `GL_POST_IMAGE_TRANSFORM_COLOR_TABLE_HP`.
format	The format in which the lookup table values are to be returned. The allowable values are `GL_LUMINANCE` and `GL_RGBA`.
type	The data type in which the lookup table value are to be returned. The allowable values are `GL_UNSIGNED_BYTE` and `GL_UNSIGNED_SHORT`.
table	Pointer to an array in which the lookup table values are returned.

Description

`glGetColorTableEXT` is part of the `EXT_color_table` extension. At present, only the subset of `EXT_color_table` needed to support the `HP_image_transform` extension has been implemented. See

`glImageTransformParameterHP` for a description of the image transformation process.

B

This routine is used to return the current contents of a color table. No pixel transfer operations are performed on the pixel values that are returned, but applicable pixel storage modes are performed. Color components that are requested in the specified format, but which are not included in the internal format of the color lookup table are returned as zero. The assignments of internal color components to the components requested by *format* are:

Internal Component	Resulting Component
red	red
green	green
blue	blue
alpha	alpha
luminance	red

Notes

The combination of a *format* of `GL_RGBA` and a *type* of `GL_UNSIGNED_SHORT` is not currently supported.

Errors

`GL_INVALID_ENUM` is generated if *target* is not one of the allowable values.

`GL_INVALID_ENUM` is generated if *format* is not one of the allowable values.

`GL_INVALID_ENUM` is generated if *type* is not one of the allowable values.

Associated Gets

`glGetColorTableParameterEXT`

See Also

`glColorTableEXT`,
`glColorTableParameterEXT`,
`glImageTransformParameterHP`.

glGetColorTableParameter*vEXT

Get color lookup table parameters.

C Specification

```
void glGetColorTableParameterfvEXT(GLenum   target,
                                   GLenum   pname,
                                   GLfloat *params)

void glGetColorTableParameterivEXT(GLenum target,
                                   GLenum pname,
                                   GLint *params)
```

Parameters

target — The target color table. Must be `GL_POST_IMAGE_TRANSFORM_COLOR_TABLE_HP`.

pname — The symbolic name of a color lookup table parameter. Must be one of:

`GL_COLOR_TABLE_SCALE_EXT`,
`GL_COLOR_TABLE_BIAS_EXT`,
`GL_COLOR_TABLE_FORMAT_EXT`,
`GL_COLOR_TABLE_WIDTH_EXT`,
`GL_COLOR_TABLE_RED_SIZE_EXT`,
`GL_COLOR_TABLE_GREEN_SIZE_EXT`,
`GL_COLOR_TABLE_BLUE_SIZE_EXT`,
`GL_COLOR_TABLE_ALPHA_SIZE_EXT`, or
`GL_COLOR_TABLE_LUMINANCE_SIZE_EXT`.

params — A pointer to an array where the values of the parameter will be stored.

Description

glGetColorTableParameterEXT is part of the EXT_color_table extension, which adds several color lookup tables to the pixel transfer path. At present, only the subset of EXT_color_table needed to support the HP_image_transform extension has been implemented.

glGetColorTableParameterEXT retrieves the color table scale and bias parameters set by glColorTableParameterEXT, as well as the format and size parameters set by glColorTableEXT. The *target* argument must be GL_POST_IMAGE_TRANSFORM_COLOR_TABLE_HP to retrieve any of the post-image transform color table parameters.

The *pname* argument must be GL_COLOR_TABLE_SCALE_EXT to retrieve the scale factors. In this case, *params* points to an array of four elements, which receive the scale factors for red, green, blue, and alpha, in that order.

The *pname* argument must be GL_COLOR_TABLE_BIAS_EXT to retrieve the bias terms, while *params* points to an array of four elements, which receive the bias terms for red, green, blue, and alpha, in that order.

The *pname* argument may also be one of the symbolic constants in the following table, in which case the specified parameter will be returned in the location indicated by *params*:

pname	**Meaning**
GL_COLOR_TABLE_FORMAT_EXT	Internal format (e.g. GL_RGBA)
GL_COLOR_TABLE_WIDTH_EXT	Number of elements in table
GL_COLOR_TABLE_RED_SIZE_EXT	Size of red component, in bits
GL_COLOR_TABLE_GREEN_SIZE_EXT	Size of green component
GL_COLOR_TABLE_BLUE_SIZE_EXT	Size of blue component
GL_COLOR_TABLE_ALPHA_SIZE_EXT	Size of alpha component
GL_COLOR_TABLE_LUMINANCE_SIZE_EXT	Size of luminance component

`glGetColorTableParameter*vEXT`

Errors

`GL_INVALID_ENUM` is generated if *target* or *pname* is not an acceptable value.

See Also

`glColorTableEXT`,
`glColorTableParameterEXT`.

glGetConvolutionFilterEXT

Get current 2D convolution filter kernel.

C Specification

```
void glGetConvolutionFilterEXT(GLenum  target,
                               GLenum  format,
                               GLenum  type,
                               GLvoid *image)
```

Parameters

target	The filter to be retrieved. Must be `GL_CONVOLUTION_2D_EXT`.
format	Format of the output image. Must be one of `GL_RGBA` or `GL_LUMINANCE`.
type	Data type of components in the output image. Must be `GL_FLOAT`.
image	Pointer to storage for the output image.

Description

`glGetConvolutionFilterEXT` returns the current 2D convolution filter kernel as an image. The one- or two-dimensional kernel is placed in *image* according to the specifications in *format* and *type*. No pixel transfer operations are performed on this image, but the relevant pixel storage modes are applied.

Color components that are present in *format* but not included in the internal format of the filter are returned as zero. The assignments of internal color components to the components of *format* are as follows:

Internal Component	Resulting Component
red	red
green	green
blue	blue
alpha	alpha
luminance	red

Errors

`GL_INVALID_ENUM` is generated if *target* is not one of the allowable values.

`GL_INVALID_ENUM` is generated if *format* is not one of the allowable values.

`GL_INVALID_ENUM` is generated if *type* is not one of the allowable values.

Associated Gets

`glGetConvolutionParameterEXT`

See Also

`glConvolutionFilter2DEXT`,
`glConvolutionParameterEXT`,
`glGetConvolutionParameterEXT`.

glGetConvolutionParameter*vEXT

Get convolution parameters.

C Specification

```
void glGetConvolutionParameterfvEXT(GLenum   target,
                                    GLenum   pname,
                                    GLfloat *params)

void glGetConvolutionParameterivEXT(GLenum target,
                                    GLenum pname,
                                    GLint *params)
```

Parameters

target The filter whose parameters are to be retrieved. Must be GL_CONVOLUTION_2D_EXT.

pname The parameter to be retrieved. Must be one of:

GL_CONVOLUTION_BORDER_MODE_EXT,
GL_CONVOLUTION_FILTER_SCALE_EXT,
GL_CONVOLUTION_FILTER_BIAS_EXT,
GL_CONVOLUTION_FORMAT_EXT,
GL_CONVOLUTION_WIDTH_EXT,
GL_CONVOLUTION_HEIGHT_EXT,
GL_MAX_CONVOLUTION_WIDTH_EXT,
GL_MAX_CONVOLUTION_HEIGHT_EXT, or
GL_CONVOLUTION_BORDER_COLOR_HP.

params Pointer to storage for the parameters to be retrieved.

Description

glGetConvolutionParameterEXT retrieves convolution parameters. The *target* argument determines which convolution filter is queried, while *pname* determines which parameter is returned:

- GL_CONVOLUTION_BORDER_MODE_EXT
  The convolution border mode. See glConvolutionParameterEXT for a list of border modes.
- GL_CONVOLUTION_FILTER_SCALE_EXT
  The current filter scale factors. The *params* argument must be a pointer to an array of four elements, which will receive the red, green, blue, and alpha filter scale factors in that order.
- GL_CONVOLUTION_FILTER_BIAS_EXT
  The current filter bias factors. The *params* argument must be a pointer to an array of four elements, which will receive the red, green, blue, and alpha filter bias terms in that order.
- GL_CONVOLUTION_FORMAT_EXT
  The current internal format. See glConvolutionFilter2DEXT for a list of allowable formats.
- GL_CONVOLUTION_WIDTH_EXT
  The current filter image width.
- GL_CONVOLUTION_HEIGHT_EXT
  The current filter image height.
- GL_MAX_CONVOLUTION_WIDTH_EXT
  The maximum acceptable filter image width.
- GL_MAX_CONVOLUTION_HEIGHT_EXT
  The maximum acceptable filter image height.
- GL_CONVOLUTION_BORDER_COLOR_HP
  The current convolution border color for the specified *target* is returned as an array of red, green, blue, and alpha values in the array specified by *params*. See glConvolutionParameterEXT for a description of the allowable values.

Errors

GL_INVALID_ENUM is generated if *target* is not one of the allowable values.

GL_INVALID_ENUM is generated if *pname* is not one of the allowable values.

See Also

glConvolutionFilter2DEXT,
glConvolutionParameterEXT,
glGetConvolutionFilterEXT.

glGetError

Return error information.

C Specification

```
GLenum glGetError(void void)
```

Description

`glGetError` returns the value of the error flag. Each detectable error is assigned a numeric code and symbolic name. When an error occurs, the error flag is set to the appropriate error code value. No other errors are recorded until `glGetError` is called, the error code is returned, and the flag is reset to `GL_NO_ERROR`. If a call to `glGetError` returns `GL_NO_ERROR`, there has been no detectable error since the last call to `glGetError`, or since initializing IVL.

To allow for distributed implementations, there may be several error flags. If any single error flag has recorded an error, the value of that flag is returned and that flag is reset to `GL_NO_ERROR` when `glGetError` is called. If more than one flag has recorded an error, `glGetError` returns and clears an arbitrary error flag value. Thus, `glGetError` should always be called in a loop, until it returns `GL_NO_ERROR`, if all error flags are to be reset. Note that, in a distributed implementation, there is no guarantee that the application will be able to process errors in the order in which they occur.

Initially, all error flags are set to `GL_NO_ERROR`.

The currently defined errors are as follows:

- `GL_NO_ERROR`
  No error has been recorded. The value of this symbolic constant is guaranteed to be zero.
- `GL_INVALID_ENUM`
  An unacceptable value is specified for an enumerated argument. The offending command is ignored, having no side effect other than to set the error flag.
- `GL_INVALID_VALUE`
  A numeric argument is out of range. The offending command is ignored, having no side effect other than to set the error flag.

- GL_INVALID_OPERATION
  The specified operation is not allowed in the current state. The offending command is ignored, having no side effect other than to set the error flag.
- GL_OUT_OF_MEMORY
  There is not enough memory left to execute the command. The state of IVL is undefined, except for the state of the error flags, after this error is recorded.
- GL_TABLE_TOO_LARGE_EXT
  The implementation cannot accommodate a table of the size requested by glColorTableEXT. The offending command is ignored, having no side effect other than to set the error flag.

When an error flag is set, results of an IVL operation are undefined only if GL_OUT_OF_MEMORY has occurred. In all other cases, the command generating the error is ignored and has no effect on the IVL state or frame buffer contents.

glGetImageTransformParameter*vHP

Get image transformation parameters.

C Specification

```
void glGetImageTransformParameterfvHP(GLenum   target,
                                       GLenum   pname,
                                       GLfloat *params)

void glGetImageTransformParameterivHP(GLenum target,
                                       GLenum pname,
                                       GLint *params)
```

Parameters

target The target image transformation. Must be `GL_IMAGE_TRANSFORM_2D_HP`.

pname The parameter to be queried. Must be one of:

`GL_IMAGE_SCALE_X_HP`,
`GL_IMAGE_SCALE_Y_HP`,
`GL_IMAGE_TRANSLATE_X_HP`,
`GL_IMAGE_TRANSLATE_Y_HP`,
`GL_IMAGE_ROTATE_ANGLE_HP`,
`GL_IMAGE_ROTATE_ORIGIN_X_HP`,
`GL_IMAGE_ROTATE_ORIGIN_Y_HP`,
`GL_IMAGE_MAG_FILTER_HP`,
`GL_IMAGE_MIN_FILTER_HP`, or
`GL_IMAGE_CUBIC_WEIGHT_HP`

params A pointer to a location where the value of the parameter will be stored.

Description

glGetImageTransformParameterHP is part of the HP_image_transform extension, which adds image scaling, rotation, translation, and window level mapping to the pixel transfer path.

glGetImageTransformParameterHP retrieves the image transformation parameters set by glImageTransformParameterHP. The *target* argument must be GL_IMAGE_TRANSFORM_2D_HP.

The *pname* argument may be one of the symbolic constants in the following table. The current value of the specified parameter will be returned in the location specified by *params*.

pname	Meaning
GL_IMAGE_SCALE_X_HP	x scaling factor
GL_IMAGE_SCALE_Y_HP	y scaling factor
GL_IMAGE_TRANSLATE_X_HP	x translation factor
GL_IMAGE_TRANSLATE_Y_HP	y translation factor
GL_IMAGE_ROTATE_ANGLE_HP	rotation angle in degrees
GL_IMAGE_ROTATE_ORIGIN_X_HP	x coordinate of rotation origin
GL_IMAGE_ROTATE_ORIGIN_Y_HP	y coordinate of rotation origin
GL_IMAGE_MIN_FILTER_HP	resampling filter for minification
GL_IMAGE_MAG_FILTER_HP	resampling filter for magnification
GL_IMAGE_CUBIC_WEIGHT_HP	cubic weight factor for cubic sampling

Errors

GL_INVALID_ENUM is generated if *target* or *pname* is not an acceptable value.

See Also

glImageTransformParameterHP.

glGetString

Returns a string describing the current IVL connection.

C Specification

```
const GLubyte * glGetString(GLenum name)
```

Parameters

name	Specifies a symbolic constant, one of GL_VENDOR, GL_RENDERER, GL_VERSION, or GL_EXTENSIONS.

Description

glGetString returns a pointer to a static string describing some aspect of the current IVL connection. The *name* argument can be one of the following:

- GL_VENDOR
  Returns the company responsible for this OpenGL implementation. This name does not change from release to release. For Hewlett-Packard the string is "HP".
- GL_RENDERER
  Returns the name of the renderer. This name is typically specific to a particular configuration of a hardware platform. It does not change from release to release.
- GL_VERSION
  Returns a version or release number.
- GL_EXTENSIONS
  Returns a list of supported extensions, separated by spaces.

Because IVL does not include queries for the performance characteristics of an implementation, it is expected that some applications will be written to recognize known platforms and will modify their usage based on known performance characteristics of these platforms. Strings GL_VENDOR and GL_RENDERER together uniquely specify a platform, and will not change from release to release. They should be used by to select platform-specific code. The format and contents of the GL_VENDOR and the GL_RENDERER strings depend on the implementation.

The GL_VERSION string begins with a version number. The version number is of the form ⟨*major_number*⟩.⟨*minor_number*⟩ or ⟨*major_number*⟩.⟨*minor_number*⟩.⟨*release_number*⟩. Vendor-specific information may follow the version number. Its format depends on the implementation, but a space always separates the version number and the vendor-specific information.

All strings are null-terminated.

Notes

If an error is generated, glGetString returns zero. The client and server may support different versions or extensions. glGetString always returns a version number or list of extensions that is compatible with both the client and server. The release number always describes the server.

Errors

GL_INVALID_ENUM is generated if *name* is not an accepted value.

glImageTransformParameter*HP

Set image transformation parameters.

C Specification (for Single-Value Attributes)

```
void glImageTransformParameterfHP(GLenum  target,
                                  GLenum  pname,
                                  GLfloat param)

void glImageTransformParameteriHP(GLenum target,
                                  GLenum pname,
                                  GLint  param)
```

Parameters

target The target image transformation. Must be `GL_IMAGE_TRANSFORM_2D_HP`.

pname The parameter to be set. Must be one of:

`GL_IMAGE_SCALE_X_HP`,
`GL_IMAGE_SCALE_Y_HP`,
`GL_IMAGE_ROTATE_ANGLE_HP`,
`GL_IMAGE_ROTATE_ORIGIN_X_HP`,
`GL_IMAGE_ROTATE_ORIGIN_Y_HP`,
`GL_IMAGE_TRANSLATE_X_HP`,
`GL_IMAGE_TRANSLATE_Y_HP`,
`GL_IMAGE_MIN_FILTER_HP`,
`GL_IMAGE_MAG_FILTER_HP`, or
`GL_IMAGE_CUBIC_WEIGHT_HP`.

param The parameter value. If *pname* is `GL_IMAGE_MAG_FILTER_HP`, *param* must be one of `GL_NEAREST`, `GL_LINEAR`, or `GL_CUBIC_HP`. If *pname* is `GL_IMAGE_MIN_FILTER_HP`, *param* must be one of `GL_NEAREST`, `GL_LINEAR`, or `GL_CUBIC_HP`. If *pname* is any of the other accepted values, the specified parameter will be set to the value of *param*.

C Specification (for Multiple-Value Attributes)

```
void glImageTransformParameterfvHP(GLenum target,
                                   GLenum pname,
                                   const GLfloat *params)

void glImageTransformParameterivHP(GLenum target,
                                   GLenum pname,
                                   const GLint *params)
```

Parameters

target The target image transformation. Must be `GL_IMAGE_TRANSFORM_2D_HP`.

pname The parameter to be set. Must be one of:

`GL_IMAGE_SCALE_X_HP`,
`GL_IMAGE_SCALE_Y_HP`,
`GL_IMAGE_ROTATE_ANGLE_HP`,
`GL_IMAGE_ROTATE_ORIGIN_X_HP`,
`GL_IMAGE_ROTATE_ORIGIN_Y_HP`,
`GL_IMAGE_TRANSLATE_X_HP`,
`GL_IMAGE_TRANSLATE_Y_HP`,
`GL_IMAGE_MIN_FILTER_HP`,
`GL_IMAGE_MAG_FILTER_HP`, or
`GL_IMAGE_CUBIC_WEIGHT_HP`.

params The parameter value. If *pname* is `GL_IMAGE_MAG_FILTER_HP`, *params* must be one of `GL_NEAREST`, `GL_LINEAR`, or `GL_CUBIC_HP`. If *pname* is `GL_IMAGE_MIN_FILTER_HP`, *params* must be one of `GL_NEAREST`, `GL_LINEAR`, or `GL_CUBIC_HP`. If *pname* is any of the other accepted values, the specified parameter will be set to the value of *params*.

glImageTransformParameter*HP

Description

glImageTransformParameterHP is part of the HP_image_transform extension, which adds image scaling, rotation, translation, and window level mapping to the pixel transfer path.

Parameter values GL_IMAGE_SCALE_X_HP and GL_IMAGE_SCALE_Y_HP establish the scaling factors. GL_IMAGE_ROTATE_ANGLE_HP sets the rotation angle to be used (in degrees), and GL_IMAGE_ROTATE_ORIGIN_X_HP and GL_IMAGE_ROTATE_ORIGIN_Y_HP specify the point about which the image is to be scaled and rotated. If the specified angle is positive, the rotation will be counterclockwise about the specified rotation origin. If the specified angle is negative, the rotation will be clockwise about the rotation origin. GL_IMAGE_TRANSLATE_X_HP and GL_IMAGE_TRANSLATE_Y_HP set the translation factors. All of these parameters (scale, rotation, translation, rotation origin) are specified in terms of the input image's coordinates, where the lower left corner of the image has coordinates of (0,0).

Notes

GL_IMAGE_MIN_FILTER_HP defines the resampling technique that is to be applied if the image is minified by the scaling factors. GL_IMAGE_MAG_FILTER_HP establishes the resampling technique that is to be used after the other image transformation operators have been applied if the image is deemed to have been magnified. GL_IMAGE_CUBIC_WEIGHT_HP defines the cubic weighting coefficient that is to be used whenever the resampling technique is set to GL_CUBIC_HP.

When enabled, the image transformation operation uses the current set of image transformation parameters to compute a new window coordinate for each incoming pixel. Although image transformation parameters are specified separately, the scaling, rotation, and translation operations are all applied simultaneously (as if the transformation was encoded in a matrix and the resulting matrix was applied to each incoming pixel coordinate). In the case of 2D image transformation, if (Rx,Ry) specifies the rotation origin, the effect of applying the 2D image transformation operators can be defined as follows. First, the image is translated by Rx in the x direction and Ry in the y direction so that its rotation origin is at the origin of the 2D coordinate system. Second, the x and y scaling factors are applied, causing the image to be scaled as specified in x and y. Third, the rotation angle is applied, causing the image to be rotated

about the origin by the specified angle. Next, the image is translated by Rx in the x direction and Ry in the y direction. Finally, the scaled and rotated image is translated by the specified translation factors. Resampling occurs after the scaling/rotation/translation operations have been applied.

Since multiple input pixels can be mapped into a single output pixel (minification of input image), or since output pixels might not have any input pixels mapped to them (magnification of input image), some method of resampling is required. If the resampling method is `GL_NEAREST`, each output pixel will have the value of the input pixel whose transformed coordinate value is nearest (in Manhattan distance). If the resampling method is `GL_LINEAR`, each output pixel will have a value that is the weighted average of the four input pixels whose transformed coordinate values are nearest.

If the resampling method is `GL_CUBIC_HP`, each output pixel will have a value that is affected by the 16 input pixels whose transformed coordinate values are nearest. The 16 input pixels will be used to perform a cubic convolution interpolation to determine the value of the output pixel. The cubic weight factor is a floating-point value that is applied to the cubic interpolation in the manner described in *Digital Image Warping* by George Wolberg. Visually pleasing cubic weighting values are typically in the range [->1,0]. The values between 1.0 and 0.5 are most commonly used. Cubic interpolation is not performed along the edges of the input image, where a neighborhood of 16 pixels is not available. As a result, the outer one pixel edge of the input image will not be transformed. For image scaling values of 1.0 in X and in Y, this causes the output image to be two pixels less than the input image in both width and height.

Defaults

The default values for X and Y scale factors are 1.0, the default value for the cubic weighting factor is 0.5, the default minification and magnification filters are `GL_NEAREST`, and all other defaults are 0. Image transformation is disabled by default, and can be enabled by calling `glEnable` with the value `GL_IMAGE_TRANSFORM_2D_HP`.

glImageTransformParameter*HP

Errors

GL_INVALID_ENUM is generated if *target* or *pname* is not an acceptable value.

GL_INVALID_ENUM is generated if *pname* is GL_IMAGE_MAG_FILTER_HP and the specified parameter value is not one of GL_NEAREST, GL_LINEAR, or GL_CUBIC_HP.

GL_INVALID_ENUM is generated if *pname* is GL_IMAGE_MIN_FILTER_HP and the specified parameter value is not one of GL_NEAREST, GL_LINEAR, or GL_CUBIC_HP.

Associated Gets

glEnable
glGetImageTransformParameterHP

glIsEnabled

Test whether a capability is enabled.

C Specification

```
GLboolean glIsEnabled(GLenum cap)
```

Parameters

cap	Specifies a symbolic constant indicating an IVL capability.

Description

`glIsEnabled` returns `GL_TRUE` if *cap* is an enabled capability and returns `GL_FALSE` otherwise. The following capabilities are accepted for *cap*:

- `GL_CONVOLUTION_2D_EXT`
  See `glConvolutionFilter2DEXT`.
- `GL_SCISSOR_TEST`
  See `glScissor`.
- `GL_POST_IMAGE_TRANSFORM_COLOR_TABLE_HP`
  If enabled, perform a color table lookup operation after the image transformation operation.
- `GL_IMAGE_TRANSFORM_2D_HP`
  If enabled, perform an image transformation operation as part of pixel transfer. See `glImageTransformParameterHP`.

Notes

If an error is generated, `glIsEnabled` returns zero.

glIsEnabled

Errors

GL_INVALID_ENUM is generated if *cap* is not an accepted value.

See Also

glConvolutionFilter2DEXT,
glEnable,
glImageTransformParameterHP,
glScissor.

glPixelStore*

Set pixel storage modes.

C Specification

```
void glPixelStoref(GLenum  pname,
                   GLfloat param)

void glPixelStorei(GLenum pname,
                   GLint  param)
```

Parameters

pname Specifies the symbolic name of the parameter to be set. Four values affect the packing of pixel data into memory:

`GL_PACK_ROW_LENGTH`,
`GL_PACK_SKIP_PIXELS`,
`GL_PACK_SKIP_ROWS`, and
`GL_PACK_ALIGNMENT`.

Four other values affect the unpacking of pixel data from memory:

`GL_UNPACK_ROW_LENGTH`,
`GL_UNPACK_SKIP_PIXELS`,
`GL_UNPACK_SKIP_ROWS`, and
`GL_UNPACK_ALIGNMENT`.

param Specifies the value to assign to *pname*.

Description

`glPixelStore` sets pixel storage modes that affect the operation of subsequent `glDrawPixels` and `glReadPixels`.

The *pname* argument is a symbolic constant indicating the parameter to be set, and *param* is the new value. Four of the eight storage parameters affect how

pixel data is returned to client memory, and are therefore significant only for glReadPixels commands. They are as follows:

- GL_PACK_ROW_LENGTH
  If greater than zero, GL_PACK_ROW_LENGTH defines the number of pixels in a row. If the first pixel of a row is placed at location p in memory, then the location of the first pixel of the next row is obtained by skipping

$$k = \begin{cases} nl & \text{if } s \geq a \\ \frac{a}{s}\left\lceil \frac{snl}{a} \right\rceil & \text{if } s < a \end{cases}$$

  components, where n is the number of components in a pixel, l is the number of pixels in a row (GL_PACK_ROW_LENGTH if it is greater than zero, the width argument to the pixel routine otherwise), a is the value of GL_PACK_ALIGNMENT, and s is the size, in bytes, of a single component (if $a<s$, then it is as if $a=s$).

  The word "component" in this description refers to the component values red, green, blue, and alpha. Storage format GL_RGBA, for example, has four components per pixel: first red, then green, then blue, and finally alpha.

- GL_PACK_SKIP_PIXELS and GL_PACK_SKIP_ROWS
  These values are provided as a convenience to the programmer; they provide no functionality that cannot be duplicated simply by incrementing the pointer passed to glReadPixels. Setting GL_PACK_SKIP_PIXELS to i is equivalent to incrementing the pointer by in components, where n is the number of components in each pixel. Setting GL_PACK_SKIP_ROWS to j is equivalent to incrementing the pointer by jk components, where k is the number of components per row, as computed above in the GL_PACK_ROW_LENGTH section.

- GL_PACK_ALIGNMENT
  Specifies the alignment requirements for the start of each pixel row in memory. The allowable values are 1 (byte-alignment), 2 (rows aligned to even-numbered bytes), 4 (word alignment), and 8 (rows start on double-word boundaries).

The other four of the eight storage parameters affect how pixel data is read from client memory. These values are significant for glDrawPixels. They are as follows:

- GL_UNPACK_ROW_LENGTH
  If greater than zero, GL_UNPACK_ROW_LENGTH defines the number of pixels in a row. If the first pixel of a row is placed at location p in memory, then the location of the first pixel of the next row is obtained by skipping

$$k = \begin{cases} nl & \text{if } s \geq a \\ \frac{a}{s}\left\lceil \frac{snl}{a} \right\rceil & \text{if } s < a \end{cases}$$

  components, where n is the number of components in a pixel, l is the number of pixels in a row (GL_UNPACK_ROW_LENGTH if it is greater than zero, the width argument to the pixel routine otherwise), a is the value of GL_UNPACK_ALIGNMENT, and s is the size, in bytes, of a single component (if $a<s$, then it is as if $a=s$).

  The word "component" in this description refers to the nonindex values red, green, blue, and alpha. Storage format GL_RGB, for example, has four components per pixel: first red, then green, then blue, and finally alpha.

- GL_UNPACK_SKIP_PIXELS and GL_UNPACK_SKIP_ROWS
  These values are provided as a convenience to the programmer; they provide no functionality that cannot be duplicated simply by incrementing the pointer passed to glDrawPixels. Setting GL_UNPACK_SKIP_PIXELS to i is equivalent to incrementing the pointer by in components or indices, where n is the number of components or indices in each pixel. Setting GL_UNPACK_SKIP_ROWS to j is equivalent to incrementing the pointer by jk components or indices, where k is the number of components or indices per row, as computed above in the GL_UNPACK_ROW_LENGTH section.

- GL_UNPACK_ALIGNMENT
  Specifies the alignment requirements for the start of each pixel row in memory. The allowable values are 1 (byte-alignment), 2 (rows aligned to even-numbered bytes), 4 (word alignment), and 8 (rows start on double-word boundaries).

The following table gives the type, initial value, and range of valid values for each of the storage parameters that can be set with glPixelStore.

pname	Type	Initial Value	Valid Range
GL_PACK_ROW_LENGTH	integer	0	[0,∞)
GL_PACK_SKIP_ROWS	integer	0	[0,∞)
GL_PACK_SKIP_PIXELS	integer	0	[0,∞)
GL_PACK_ALIGNMENT	integer	4	1, 2, 4, or 8
GL_UNPACK_ROW_LENGTH	integer	0	[0,∞)
GL_UNPACK_SKIP_ROWS	integer	0	[0,∞)
GL_UNPACK_SKIP_PIXELS	integer	0	[0,∞)
GL_UNPACK_ALIGNMENT	integer	4	1, 2, 4, or 8

glPixelStoref can be used to set any pixel store parameter. Likewise, glPixelStorei can also be used to set any of the pixel store parameters.

Errors

GL_INVALID_ENUM is generated if *pname* is not an accepted value.

GL_INVALID_VALUE is generated if a negative row length, pixel skip, or row skip value is specified, or if alignment is specified as other than 1, 2, 4, or 8.

Associated Gets

glGet (GL_PACK_ROW_LENGTH)
glGet (GL_PACK_SKIP_ROWS)
glGet (GL_PACK_SKIP_PIXELS)
glGet (GL_PACK_ALIGNMENT)
glGet (GL_UNPACK_ROW_LENGTH)
glGet (GL_UNPACK_SKIP_ROWS)
glGet (GL_UNPACK_SKIP_PIXELS)
glGet (GL_UNPACK_ALIGNMENT)

See Also

glDrawPixels,
glReadPixels.

glPixelTransfer*

Set pixel transfer modes.

C Specification

```
void glPixelTransferf(GLenum  pname,
                      GLfloat param)

void glPixelTransferi(GLenum pname,
                      GLint  param)
```

Parameters

pname Specifies the symbolic name of the pixel transfer parameter to be set. Must be one of the following:

GL_POST_CONVOLUTION_RED_SCALE_EXT,
GL_POST_CONVOLUTION_RED_BIAS_EXT,
GL_POST_CONVOLUTION_GREEN_SCALE_EXT,
GL_POST_CONVOLUTION_GREEN_BIAS_EXT,
GL_POST_CONVOLUTION_BLUE_SCALE_EXT,
GL_POST_CONVOLUTION_BLUE_BIAS_EXT,
GL_POST_CONVOLUTION_ALPHA_SCALE_EXT, or
GL_POST_CONVOLUTION_ALPHA_BIAS_EXT

param Specifies the value that *pname* is set to.

Description

glPixelTransfer sets pixel transfer modes that affect the operation of subsequent glCopyPixels, glDrawPixels, and glReadPixels commands. The algorithms that are specified by pixel transfer modes operate on pixels after they are read from the frame buffer (glCopyPixels and glReadPixels) or unpacked from client memory (glDrawPixels). Pixel transfer operations occur in the same order, and in the same manner, regardless of the command that resulted in the pixel operation. Pixel storage modes (see glPixelStore) control the unpacking of pixels being read from client memory, and the packing of pixels being written back into client memory.

`glPixelTransfer*`

Pixel transfer operations are used to modify the values of pixels as they are transferred by a copy, read, or store operation. Color pixels are made up of four floating-point values with unspecified mantissa and exponent sizes, scaled such that 0.0 represents zero intensity and 1.0 represents full intensity.

In the first release of IVL, the only pixel transfer operation that is defined is the post-convolution scale and bias. When convolution is enabled, each of the four color components is multiplied by a scale factor, then added to a bias factor. These values are applied after the convolution operation, if convolution is enabled.

That is, the red component is multiplied by `GL_POST_CONVOLUTION_RED_SCALE_EXT`, then added to `GL_POST_CONVOLUTION_RED_BIAS_EXT`; the green component is multiplied by `GL_POST_CONVOLUTION_GREEN_SCALE_EXT`, then added to `GL_POST_CONVOLUTION_GREEN_BIAS_EXT`; the blue component is multiplied by `GL_POST_CONVOLUTION_BLUE_SCALE_EXT`, then added to `GL_POST_CONVOLUTION_BLUE_BIAS_EXT`; and the alpha component is multiplied by `GL_POST_CONVOLUTION_ALPHA_SCALE_EXT`, then added to `GL_POST_CONVOLUTION_ALPHA_BIAS_EXT`.

After all four color components are scaled and biased, each is clamped to the range [0,1]. The post-convolution scale and bias operation is applied immediately after the convolution operation is performed.

The following table gives the type, initial value, and range of valid values for each of the pixel transfer parameters that are set with `glPixelTransfer`.

pname	**Type**	**Initial Value**	**Valid Range**
`GL_POST_CONVOLUTION_RED_SCALE_EXT`	float	1.0	$(-\infty, \infty)$
`GL_POST_CONVOLUTION_GREEN_SCALE_EXT`	float	1.	$(-\infty, \infty)$
`GL_POST_CONVOLUTION_BLUE_SCALE_EXT`	float	1.	$(-\infty, \infty)$
`GL_POST_CONVOLUTION_ALPHA_SCALE_EXT`	float	1.	$(-\infty, \infty)$
`GL_POST_CONVOLUTION_RED_BIAS_EXT`	float	0.	$(-\infty, \infty)$
`GL_POST_CONVOLUTION_GREEN_BIAS_EXT`	float	0.	$(-\infty, \infty)$
`GL_POST_CONVOLUTION_BLUE_BIAS_EXT`	float	0.	$(-\infty, \infty)$
`GL_POST_CONVOLUTION_ALPHA_BIAS_EXT`	float	0.	$(-\infty, \infty)$

glPixelTransferi can be used to set any of the pixel transfer parameters. The *param* argument is converted to floating point before being assigned to real-valued parameters.

Errors

GL_INVALID_ENUM is generated if *pname* is not an accepted value.

Associated Gets

glGet (GL_POST_CONVOLUTION_RED_SCALE_EXT)
glGet (GL_POST_CONVOLUTION_RED_BIAS_EXT)
glGet (GL_POST_CONVOLUTION_GREEN_SCALE_EXT)
glGet (GL_POST_CONVOLUTION_GREEN_BIAS_EXT)
glGet (GL_POST_CONVOLUTION_BLUE_SCALE_EXT)
glGet (GL_POST_CONVOLUTION_BLUE_BIAS_EXT)
glGet (GL_POST_CONVOLUTION_ALPHA_SCALE_EXT)
glGet (GL_POST_CONVOLUTION_ALPHA_BIAS_EXT)

See Also

glCopyPixels,
glDrawPixels,
glPixelStore,
glReadPixels.

glRasterPos*

Specify the raster position for pixel operations.

C Specification (Discrete Coordinate Values)

```
void glRasterPos2i(GLint x,
                   GLint y)
```

Parameters

x, y	Specify the x and y object coordinates (if present) for the raster position.

C Specification (Vector of Coordinate Values)

```
void glRasterPos2iv(const GLint *v)
```

Parameters

v	Specifies a pointer to an array of two elements, specifying x and y coordinates, respectively.

Description

IVL maintains the current position in window coordinates. This position, called the raster position, is maintained with subpixel accuracy. It is used to position pixel write operations. See `glDrawPixels` and `glCopyPixels`.

The current raster position contains two window coordinates (x and y), an eye coordinate distance, and a valid bit.

B

`glRasterPos2` uses the argument values for x and y.

The object coordinates presented by `glRasterPos` are transformed by the current model view and projection matrices and passed to the clipping stage. The first release of IVL does not implement the model view and projection matrices, so the identity matrix is used by default. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position,

and the GL_CURRENT_RASTER_POSITION_VALID flag is set. If the vertex is culled, then the valid bit is cleared and the current raster position and its associated data are undefined.

Defaults

Initially, the current raster position has (x,y,z,w) set to (0,0,0,1), and the valid bit is set.

Notes

The raster position is modified by glRasterPos. When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to IVL state).

Associated Gets

glGet (GL_CURRENT_RASTER_POSITION)
glGet (GL_CURRENT_RASTER_POSITION_VALID)

See Also

glCopyPixels,
glDrawPixels.

glReadBuffer

Select a color buffer source for pixels.

C Specification

```
void glReadBuffer(GLenum mode)
```

Parameters

mode	Specifies a color buffer. Accepted values are `GL_FRONT_LEFT`, `GL_BACK_LEFT`, `GL_FRONT`, and `GL_BACK`.

Description

`glReadBuffer` specifies a color buffer as the source for subsequent `glCopyPixels` and `glReadPixels` commands. The *mode* argument can be one of four predefined values. In a fully configured system, `GL_FRONT` and `GL_FRONT_LEFT` name the front left buffer, and `GL_BACK_LEFT` and `GL_BACK` name the back left buffer.

Non-stereo double-buffered configurations have only a front left and a back left buffer. Single-buffered configurations have a front left and a front right buffer if stereo, and only a front left buffer if the output device does not support stereo. It is an error to specify a nonexistent buffer to `glReadBuffer`.

Defaults

By default, *mode* is `GL_FRONT` in single-buffered configurations, and `GL_BACK` in double-buffered configurations.

B

Notes

Stereo is not supported in the initial release of IVL.

Errors

GL_INVALID_ENUM is generated if *mode* is not one of the accepted values.

GL_INVALID_OPERATION is generated if *mode* specifies a buffer that does not exist.

Associated Gets

glGet (GL_READ_BUFFER)

See Also

glCopyPixels,
glDrawBuffer,
glReadPixels.

glReadPixels

Read a block of pixels from the frame buffer.

C Specification

```
void glReadPixels(GLint   x,
                  GLint   y,
                  GLsizei width,
                  GLsizei height,
                  GLenum  format,
                  GLenum  type,
                  GLvoid *pixels)
```

Parameters

x, *y*	Specify the window coordinates of the first pixel that is read from the frame buffer. This location is the lower left corner of a rectangular block of pixels.
width, *height*	Specify the dimensions of the pixel rectangle. If *width* and *height* are both set to a value of one (`1`), this corresponds to a single pixel.
format	Specifies the format of the pixel data. The following symbolic values are accepted: `GL_RGBA` and `GL_LUMINANCE`.
type	Specifies the data type of the pixel data. Must be one of `GL_UNSIGNED_BYTE` or `GL_UNSIGNED_SHORT`.
pixels	Returns the pixel data.

B

Description

`glReadPixels` returns pixel data from the frame buffer, starting with the pixel whose lower left corner is at location (*x*, *y*), and stores the data in client memory starting at the location pointed to by *pixels*. It is the responsibility of the application to allocate space for holding the returned pixel values.

glReadPixels returns values from each pixel with lower left-hand corner at ($x+i$, $y+j$) for $0<i<width$ and $0<j<height$. This pixel is said to be the ith pixel in the jth row. Pixels are returned in row order from the lowest to the highest row, left to right across each row.

The *format* argument specifies the format for the returned pixel values. Accepted values for *format* are as follows:

GL_RGBA
GL_LUMINANCE Pixel values are read from the color buffer selected by glReadBuffer. Each color component is converted to a floating-point value such that zero intensity maps to 0.0 and full intensity maps to 1.0.

GL_LUMINANCE computes a single component value as the sum of the red, green, and blue components. The final values are clamped to the range [0,1].

In order to return pixel values as unsigned bytes or unsigned shorts, each component, c, is multiplied by the multiplier, as shown in the following table:

type	**Component Conversion**
GL_UNSIGNED_BYTE	$(2^8-1)c$
GL_UNSIGNED_SHORT	$(2^{16}-1)c$

If pixel transfer operations such as convolution and image transformation are disabled, return values are placed in memory as follows. If *format* is GL_LUMINANCE, a single value is returned and the data for the ith pixel in the jth row is placed in location (j) $>width+i$. GL_RGBA returns four values, with all values corresponding to a single pixel occupying contiguous space in *pixels*. Storage parameters set by glPixelStore, such as GL_PACK_SKIP_ROWS and GL_PACK_ROW_LENGTH, affect the way that data is written into memory. See glPixelStore for a description.

Notes

For this release, the combination of *format* `GL_RGBA` and *type* `GL_UNSIGNED_SHORT` is not supported.

Values for pixels that lie outside the window connected to the current GL context are undefined.

If an error is generated, no change is made to the contents of *pixels*.

Errors

`GL_INVALID_ENUM` is generated if *format* or *type* is not an accepted value.

`GL_INVALID_VALUE` is generated if either *width* or *height* is negative.

See Also

`glCopyPixels`,
`glDrawPixels`,
`glPixelStore`,
`glReadBuffer`.

glScissor

Define the scissor box.

C Specification

```
void glScissor(GLint   x,
               GLint   y,
               GLsizei width,
               GLsizei height)
```

Parameters

x, *y*	Specify the lower left corner of the scissor box. Initially (0,0).
width, *height*	Specify the width and height of the scissor box. When an IVL context is first attached to a window, *width* and *height* are set to the dimensions of that window.

Description

The `glScissor` routine defines a rectangle, called the scissor box, in window coordinates. The first two arguments, *x* and *y*, specify the lower left corner of the box. *width* and *height* specify the width and height of the box.

The scissor test is enabled and disabled using `glEnable` and `glDisable` with argument `GL_SCISSOR_TEST`. While the scissor test is enabled, only pixels that lie within the scissor box can be modified by drawing commands. Window coordinates have integer values at the shared corners of frame buffer pixels, so `glScissor(0,0,1,1)` allows only the lower left pixel in the window to be modified, and `glScissor(0,0,0,0)` disallows modification to all pixels in the window.

When the scissor test is disabled, it is as though the scissor box includes the entire window.

Errors

GL_INVALID_VALUE is generated if either *width* or *height* is negative.

Associated Gets

glGet (GL_SCISSOR_BOX)
glIsEnabled (GL_SCISSOR_TEST)

See Also

glDisable,
glEnable.

glX.Intro

Introduction to IVL in the X window system.

Overview

The Image Visualization Library (IVL) is a high-performance 2-D-oriented renderer. It is available in the X window system through the GLX extension. (The initial IVL implementation includes a subset of the GLX calls defined for use with the OpenGL API.) Use `glXQueryExtension` to establish whether the GLX extension is supported by an X server.

GLX extended servers make a subset of their visuals available for IVL rendering. Drawables created with these visuals can also be rendered using the core X renderer and with the renderer of any other X extension that is compatible with all core X visuals.

GLX extends drawables with several buffers other than the standard color buffer. These buffers include back and auxiliary color buffers, a depth buffer, a stencil buffer, and a color accumulation buffer. Some or all are included in each X visual that supports IVL.

To render using IVL into an X drawable, you must first choose a visual that defines the required IVL buffers. `glXChooseVisual` can be used to simplify selecting a compatible visual. If more control of the selection process is required, use `XGetVisualInfo` and `glXGetConfig` to select among all the available visuals.

Use the selected visual to create both a GLX context and an X drawable. GLX contexts are created with `glXCreateContext`, and drawables are created with either `XCreateWindow` or `glXCreateGLXPixmap`. Finally, bind the context and the drawable together using `glXMakeCurrent`. This context/drawable pair becomes the current context and current drawable, and it is used by all IVL commands until `glXMakeCurrent` is called with different arguments.

Both core X and IVL commands can be used to operate on the current drawable. The X and IVL command streams are not synchronized, however, except at explicitly created boundaries generated by calling `XSync`, and `glFlush`.

Notes

A color map must be created and passed to `XCreateWindow`. See the example code above.

A GLX context must be created and attached to an X drawable before IVL commands can be executed. IVL commands issued while no context/drawable pair is current are ignored.

Exposure events indicate that *all* buffers associated with the specified window may be damaged and should be repainted. Although certain buffers of some visuals on some systems may never require repainting (the depth buffer, for example), it is incorrect to code assuming that these buffers will not be damaged.

GLX commands manipulate `XVisualInfo` structures rather than pointers to visuals or visual IDs. `XVisualInfo` structures contain `visual`, `visual ID`, `screen`, and `depth` elements, as well as other X-specific information.

IVL does not support the use of all ancillary buffer types. For example, it does not support auxiliary color buffers, a depth buffer, a stencil buffer, or a color accumulation buffer.

Examples

Below is the minimum code required to create an RGBA-format, IVL- compatible X window and clear it to yellow. The code is correct, but it does not include any error checking. Return values `dpy`, `vi`, `cx`, `cmap`, and `win` should all be tested.

```
#include <GL/glx.h>
#include <GL/gl.h>
#include <unistd.h>

static int attributeListSgl[]
  = { GLX_RGBA, GLX_RED_SIZE, 1,
      /* Get the deepest buffer with one red bit. */
      GLX_GREEN_SIZE, 1, GLX_BLUE_SIZE, 1, None
    };

static int attributeListDbl[]
  = { GLX_RGBA, GLX_DOUBLE_BUFFER,
      /* In case single-buffering is not supported. */
      GLX_RED_SIZE, 1, GLX_GREEN_SIZE, 1, GLX_BLUE_SIZE, 1, None
    };

static Bool WaitForNotify (Display *d, XEvent *e, char *arg)
```

```
   {
    return (e->type == MapNotify) && (e->xmap.window == (Window)arg);
   }

int main (int argc, char ** argv)

   {
    Display * dpy;
    XVisualInfo * vis;
    Colormap cmap;
    XSetWindowAttributes swa;
    Window win;
    GLXContext ctx;
    XEvent event;
    int swap_flag = FALSE;

    /* Get a connection. */
    dpy = XOpenDisplay(0);

    /* Get an appropriate visual. */
    vis = ``glXChooseVisual''(dpy, DefaultScreen(dpy),
                        attributeListSgl);
    if (vis == NULL)
       {
        vis = ``glXChooseVisual''(dpy, DefaultScreen(dpy),
                            attributeListDbl);
        swap_flag = TRUE;
       }

    /* Create a GLX context. */
    ctx = ``glXCreateContext''(dpy, vis, 0, GL_TRUE);

    /* Create a color map. */
    cmap = XCreateColormap(dpy, RootWindow(dpy, vis->screen),
                           vis->visual, AllocNone);

    /* Create a window. */
    swa.colormap = cmap;
    swa.border_pixel = 0;
    swa.event_mask = StructureNotifyMask;
    win = XCreateWindow(dpy, RootWindow(dpy, vis->screen),
                        0, 0, 100, 100, 0, vis->depth,
                        InputOutput, vis->visual,
                        CWBorderPixel|CWColormap|CWEventMask,
                        &swa);
    XMapWindow (dpy, win);

    XIfEvent (dpy, &event, WaitForNotify, (char *)win);
```

```
            /* Connect the context to the window. */
            ``glXMakeCurrent'' (dpy, win, ctx);

            /* Clear the buffer. */
            ``glClearColor'' (1, 1, 0, 1);
            ``glClear'' (GL_COLOR_BUFFER_BIT);
            ``glFlush'' ();
            if (swap_flag)
               ``glXSwapBuffers'' (dpy, win);

            /* Wait a while. */
            sleep (10);
          }
```

See Also

glXCreateContext,
glXCreateGLXPixmap,
glXDestroyContext,
glXGetConfig,
glXSwapBuffers,
XCreateColormap,
XCreateWindow,
XSync.

glXChooseVisual

Return a visual that matches specified attributes.

C Specification

```
XVisualInfo* glXChooseVisual(Display *dpy,
                             int      screen,
                             int     *attribList)
```

Parameters

dpy	Specifies the connection to the X server.
screen	Specifies the screen number.
attribList	Specifies a list of Boolean attributes and integer attribute/value pairs. The last attribute must be `None`.

Description

`glXChooseVisual` returns a pointer to an XVisualInfo structure describing the visual that best meets a minimum specification. The Boolean GLX attributes of the visual that is returned will match the specified values, and the integer GLX attributes will meet or exceed the specified minimum values. If all other attributes are equivalent, then TrueColor and PseudoColor visuals have priority over DirectColor and StaticColor visuals, respectively. If no conforming visual exists, `NULL` is returned. To free the data returned by this function, use `XFree`.

The interpretations of the various GLX visual attributes are as follows:

- `GLX_LEVEL`
  Must be followed by an integer buffer-level specification. This specification is honored exactly. Buffer level zero corresponds to the default frame buffer of the display. Buffer level one is the first overlay frame buffer, level two the second overlay frame buffer, and so on. Negative buffer levels correspond to underlay frame buffers.
- `GLX_RGBA`
  If present, only visuals that support RGBA rendering are considered.

- GLX_DOUBLEBUFFER
  If present, only double-buffered visuals are considered. Otherwise, only single-buffered visuals are considered.
- GLX_RED_SIZE
  Must be followed by a non-negative minimum size specification. If this value is zero, the smallest available red buffer is preferred. Otherwise, the largest available red buffer of at least the minimum size is preferred.
- GLX_GREEN_SIZE
  Must be followed by a non-negative minimum size specification. If this value is zero, the smallest available green buffer is preferred. Otherwise, the largest available green buffer of at least the minimum size is preferred.
- GLX_BLUE_SIZE
  Must be followed by a non-negative minimum size specification. If this value is zero, the smallest available blue buffer is preferred. Otherwise, the largest available blue buffer of at least the minimum size is preferred.
- GLX_ALPHA_SIZE
  Must be followed by a non-negative minimum size specification. If this value is zero, the smallest available alpha buffer is preferred. Otherwise, the largest available alpha buffer of at least the minimum size is preferred.
- GLX_X_VISUAL_TYPE_EXT
  Must be followed by an X visual type (one of `TrueColor`, `DirectColor`, `PseudoColor`, `StaticColor`, `GrayScale`, or `StaticGray`). If present, only visuals of the specified type will be considered.

Defaults

All Boolean GLX attributes default to `False`. All integer GLX attributes default to zero. Default specifications are superseded by attributes included in *attribList*. Boolean attributes included in *attribList* are understood to be `True`. Integer attributes are followed immediately by the corresponding desired or minimum value. The list must be terminated with `None`.

Examples

```
attribList = {GLX_RGBA, GLX_RED_SIZE, 4, GLX_GREEN_SIZE, 4,
              GLX_BLUE_SIZE, 4, None};
```

Specifies a single-buffered RGBA visual in the normal frame buffer, not an overlay or underlay buffer. The returned visual supports at least four bits each of red, green, and blue, and possibly no bits of alpha. It does not support double-buffering.

Notes

`XVisualInfo` is defined in `Xutil.h`. It is a structure that includes visual, visual ID, screen, and depth elements.

`glXChooseVisual` is implemented as a client-side utility using only XGetVisual-Info and `glXGetConfig`. Calls to these two routines can be used to implement selection algorithms other than the generic one implemented by `glXChooseVisual`.

GLX implementors are strongly discouraged, but not prohibited, from changing the selection algorithm used by `glXChooseVisual`. Therefore, selections may change from release to release of the client-side library.

There is no direct filter for picking only visuals that support GLXPixmaps. GLXPixmaps are supported for visuals whose `GLX_BUFFER_SIZE` is one of the pixmap depths supported by the X server.

The first release of IVL only supports visuals for RGBA rendering.

Errors

`NULL` is returned if an undefined GLX attribute is encountered in *attribList*.

See Also

`glXCreateContext`,
`glXGetConfig`,
`XGetVisualInfo` (for X, not GLX, visuals).

glXCreateContext

Create a new GLX rendering context.

C Specification

```
GLXContext glXCreateContext(Display     *dpy,
                            XVisualInfo *vis,
                            GLXContext   shareList,
                            Bool         direct)
```

Parameters

dpy	Specifies the connection to the X server.
vis	Specifies the visual that defines the frame buffer resources available to the rendering context. It is a pointer to an XVisualInfo structure, not a visual ID or a pointer to a Visual.
shareList	Specifies the context with which to share display lists. `NULL` indicates that no sharing is to take place. *shareList* must be set to `NULL` for this release.
direct	Specifies whether rendering is to be done with a direct connection to the graphics system if possible (`True`) or through the X server (`False`). *direct* must be set to `TRUE` for this release.

Description

B

`glXCreateContext` creates a GLX rendering context and returns its handle. This context can be used to render into both windows and GLX pixmaps. If `glXCreateContext` fails to create a rendering context, `NULL` is returned.

If *direct* is `True`, then a direct rendering context is created if the implementation supports direct rendering and the connection is to an X server that is local. If *direct* is `False`, then a rendering context that renders through the X server is always created. Direct rendering provides a performance advantage in some

implementations. However, direct rendering contexts cannot be shared outside a single process, and they may not support rendering to GLX pixmaps.

Notes

`XVisualInfo` is defined in `Xutil.h`. It is a structure that includes visual, visual ID, screen, and depth elements.

A process is a single execution environment, implemented in a single address space, consisting of one or more threads. A thread is one of a set of subprocesses that share a single address space, but maintain separate program counters, stack spaces, and other related `global` data. A thread that is the only member of its subprocess group is equivalent to a process.

Errors

`NULL` is returned if execution fails on the client side.

`BadValue` is generated if *vis* is not a valid visual (e.g., if the GLX implementation does not support it).

`BadAlloc` is generated if the server does not have enough resources to allocate the new context.

See Also

`glXDestroyContext`,
`glXGetConfig`,
`glXMakeCurrent`.

glXCreateGLXPixmap

Create an off-screen GLX rendering area.

C Specification

```
GLXPixmap glXCreateGLXPixmap(Display     *dpy,
                             XVisualInfo *vis,
                             Pixmap       pixmap)
```

Parameters

dpy	Specifies the connection to the X server.
vis	Specifies the visual that defines the structure of the rendering area. It is a pointer to an XVisualInfo structure, not a visual ID or a pointer to a Visual.
pixmap	Specifies the X pixmap that will be used as the front left color buffer of the off-screen rendering area.

Description

`glXCreateGLXPixmap` creates an off-screen rendering area and returns its XID. Any GLX rendering context that was created with respect to *vis* can be used to render into this off-screen area. Use `glXMakeCurrent` to associate the rendering area with a GLX rendering context.

The X pixmap identified by *pixmap* is used as the front left buffer of the resulting off-screen rendering area. All other buffers specified by *vis*, including color buffers other than the front left buffer, are created without externally visible names. GLX pixmaps with double-buffering are supported. However, `glXSwapBuffers` does not affect these pixmaps.

Direct rendering contexts may not be able to be used to render into GLX pixmaps.

Notes

`XVisualInfo` is defined in `Xutil.h`. It is a structure that includes visual, visual ID, screen, and depth elements.

Errors

`BadMatch` is generated if the depth of *pixmap* does not match the `GLX_BUFFER_SIZE` value of *vis*, or if *pixmap* was not created with respect to the same screen as *vis*.

`BadValue` is generated if *vis* is not a valid XVisualInfo pointer (e.g., if the GLX implementation does not support this visual).

`BadPixmap` is generated if *pixmap* is not a valid pixmap.

`BadAlloc` is generated if the server cannot allocate the GLX pixmap.

See Also

`glXCreateContext`,
`glXDestroyGLXPixmap`,
`glXMakeCurrent`,
`glXSwapBuffers`.

glXDestroyContext

Destroy a GLX context.

C Specification

```
void glXDestroyContext(Display   *dpy,
                       GLXContext ctx)
```

Parameters

dpy	Specifies the connection to the X server.
ctx	Specifies the GLX context to be destroyed.

Description

If the specified GLX rendering context *ctx* is not current to any thread, `glXDestroyContext` destroys it immediately. Otherwise, *ctx* is destroyed when it is no longer current to any thread. In either case, the resource ID referenced by *ctx* is freed immediately.

Errors

`GLX_BAD_CONTEXT` is generated if *ctx* is not a valid GLX context.

See Also

`glXCreateContext`,
`glXMakeCurrent`.

glXDestroyGLXPixmap

Destroy a GLX pixmap.

C Specification

```
void glXDestroyGLXPixmap(Display  *dpy,
                         GLXPixmap pix)
```

Parameters

dpy	Specifies the connection to the X server.
pix	Specifies the GLX pixmap to be destroyed.

Description

If the specified GLX pixmap, *pix*, is not current to any client, `glXDestroyGLX-Pixmap` destroys it immediately. Otherwise, *pix* is destroyed when it is no longer current to any client. In either case, the resource ID is freed immediately.

Errors

`GLX_BAD_PIXMAP` is generated if *pix* is not a valid GLX pixmap.

See Also

`glXCreateGLXPixmap`,
`glXMakeCurrent`.

glXGetConfig

Return information about GLX visuals.

C Specification

```
int glXGetConfig(Display     *dpy,
                 XVisualInfo *vis,
                 int          attrib,
                 int         *value)
```

Parameters

dpy	Specifies the connection to the X server.
vis	Specifies the visual to be queried. It is a pointer to an XVisualInfo structure, not a visual ID or a pointer to a Visual.
attrib	Specifies the visual attribute to be returned.
value	Returns the requested value.

Description

`glXGetConfig` sets *value* to the *attrib* value of windows or GLX pixmaps created with respect to *vis*. `glXGetConfig` returns an error code if it fails for any reason. Otherwise, zero is returned.

attrib is one of the following:

- `GLX_USE_GL`
  `True` if IVL rendering is supported by this visual, `False` otherwise.
- `GLX_BUFFER_SIZE`
  Number of bits per color buffer. For visuals that support RGBA rendering, `GLX_BUFFER_SIZE` is the sum of `GLX_RED_SIZE`, `GLX_GREEN_SIZE`, `GLX_BLUE_SIZE`, and `GLX_ALPHA_SIZE`.
- `GLX_LEVEL`
  Frame buffer level of the visual. Level zero is the default frame buffer. Positive levels correspond to frame buffers that overlay the default buffer, and negative levels correspond to frame buffers that underlay the default buffer.

- GLX_RGBA
  True if the visual supports RGBA rendering, False otherwise.
- GLX_DOUBLEBUFFER
  True if color buffers exist in front/back pairs that can be swapped, False otherwise.
- GLX_RED_SIZE
  Number of bits of red stored in each color buffer. Undefined if GLX_RGBA is False.
- GLX_GREEN_SIZE
  Number of bits of green stored in each color buffer. Undefined if GLX_RGBA is False.
- GLX_BLUE_SIZE
  Number of bits of blue stored in each color buffer. Undefined if GLX_RGBA is False.
- GLX_ALPHA_SIZE
  Number of bits of alpha stored in each color buffer. Undefined if GLX_RGBA is False.
- GLX_X_VISUAL_TYPE_EXT
  The name of the X visual type for this visual.

The X protocol allows a single visual ID to be instantiated with different numbers of bits per pixel. Windows or GLX pixmaps that will be rendered with IVL, however, must be instantiated with a color buffer depth of GLX_BUFFER_SIZE.

Applications are best written to select the visual that most closely meets their requirements. Creating windows or GLX pixmaps with unnecessary buffers can result in reduced rendering performance as well as poor resource allocation.

Notes

XVisualInfo is defined in Xutil.h. It is a structure that includes visual, visual ID, screen, and depth elements.

Errors

GLX_NO_EXTENSION is returned if *dpy* does not support the GLX extension.

GLX_BAD_SCREEN is returned if the screen of *vis* does not correspond to a screen.

GLX_BAD_ATTRIB is returned if *attrib* is not a valid GLX attribute.

GLX_BAD_VISUAL is returned if *vis* doesn't support GLX and an attribute other than GLX_USE_GL is requested.

See Also

glXChooseVisual,
glXCreateContext.

glXMakeCurrent

Attach a GLX context to a window or a GLX pixmap.

C Specification

```
Bool glXMakeCurrent(Display    *dpy,
                    GLXDrawable drawable,
                    GLXContext  ctx)
```

Parameters

dpy	Specifies the connection to the X server.
drawable	Specifies a GLX drawable. Must be either an X window ID or a GLX pixmap ID.
ctx	Specifies a GLX rendering context that is to be attached to *drawable*.

Description

`glXMakeCurrent` performs two actions: It makes *ctx* the current GLX rendering context of the calling thread, replacing the previously current context if there was one, and it attaches *ctx* to a GLX drawable, either a window or a GLX pixmap. As a result of these two actions, subsequent IVL rendering calls use the rendering context, *ctx*, to modify the GLX drawable, *drawable*. Because `glXMakeCurrent` always replaces the current rendering context with *ctx*, there can be only one current context per thread.

Pending commands to the previous context, if any, are flushed before it is released.

The first time *ctx* is made current to any thread, its viewport is set to the full size of *drawable*. Subsequent calls by any thread to `glXMakeCurrent` with *ctx* have no effect on its viewport.

To release the current context without assigning a new one, call `glXMakeCurrent` with *drawable* set to `None`, and *ctx* set to `NULL`.

`glXMakeCurrent` returns `True` if it is successful, `False` otherwise. If `False` is returned, the previously current rendering context and drawable (if any) remain unchanged.

Notes

A process is a single-execution environment, implemented in a single address space, consisting of one or more threads. A thread is one of a set of subprocesses that share a single address space, but maintain separate program counters, stack spaces, and other related `global` data. A thread that is the only member of its subprocess group is equivalent to a process.

Errors

`BadMatch` is generated if *drawable* was not created with the same X screen and visual as *ctx*. It is also generated if *drawable* is `None` and *ctx* is not `None`.

`BadAccess` is generated if *ctx* was current to another thread at the time `glXMakeCurrent` was called.

`GLX_BAD_DRAWABLE` is generated if *drawable* is not a valid GLX drawable.

`GLX_BAD_CONTEXT` is generated if *ctx* is not a valid GLX context.

`GLX_BAD_CURRENT_WINDOW` is generated if there are pending IVL commands for the previous context and the current drawable is a window that is no longer valid.

`BadAlloc` may be generated in cases where the server has delayed allocation of ancillary buffers until `glXMakeCurrent` is called, only to find that it has insufficient resources to complete the allocation.

See Also

`glXCreateContext`,
`glXCreateGLXPixmap`.

glXQueryExtension

Indicate whether the GLX extension is supported.

C Specification

```
Bool glXQueryExtension(Display *dpy,
                       int     *errorBase,
                       int     *eventBase)
```

Parameters

dpy	Specifies the connection to the X server.
errorBase	Returns the base error code of the GLX server extension.
eventBase	Returns the base event code of the GLX server extension.

Description

`glXQueryExtension` returns `True` if the X server of connection dpy supports the GLX extension, `False` otherwise. If `True` is returned, then *errorBase* and *eventBase* return the error base and event base of the GLX extension. Otherwise, *errorBase* and *eventBase* are unchanged.

errorBase and *eventBase* do not return values if they are specified as `NULL`.

Notes

The *eventBase* argument is included for future extensions. GLX does not currently define any events.

glXSwapBuffers

Exchange front and back buffers.

C Specification

```
void glXSwapBuffers(Display    *dpy,
                    GLXDrawable drawable)
```

Parameters

dpy	Specifies the connection to the X server.
drawable	Specifies the drawable whose buffers are to be swapped.

Description

`glXSwapBuffers` promotes the contents of the back buffer of *drawable* to become the contents of the front buffer of *drawable*. The contents of the back buffer then become undefined. The update typically takes place during the vertical retrace of the monitor, rather than immediately after `glXSwapBuffers` is called. All GLX rendering contexts share the same notion of which are front buffers and which are back buffers.

`glXSwapBuffers` performs an implicit `glFlush` before returning. Subsequent IVL commands can be issued immediately after calling `glXSwapBuffers`, but are not executed until after the buffer exchange completes.

If *drawable* was not created with respect to a double-buffered visual, `glXSwapBuffers` has no effect, and no error is generated.

B

Notes

Synchronization of multiple GLX contexts rendering to the same double-buffered window is the responsibility of the clients. The X Synchronization Extension can be used to facilitate such cooperation.

Errors

GLX_BAD_DRAWABLE is generated if *drawable* is not a valid GLX drawable.

GLX_BAD_CURRENT_WINDOW is generated if *dpy* and *drawable* are respectively the display and drawable associated with the current context of the calling thread/process, and drawable identifies a window that is no longer valid.

See Also

glFlush.

Glossary

API
Application Programming Interface. The API is the set of subroutine calls that is available for use by an application programmer.

Back Buffer
The undisplayed buffer in a double-buffer pair. See **double-buffering**; compare with **front buffer**.

Bitplane
A rectangular array of bits mapped one-to-one with pixels. The frame buffer is a stack of bitplanes.

Client State
IVL state that is not stored in the rendering context. For example, the current drawable and the current rendering context.

Clipping
Eliminating (e.g., not displaying) some portion of the image that is being rendered. For example, images may be clipped because some portions fall outside the current window or current **scissor box**.

Color Buffer
A logical set of bitplanes. The color buffer may consist of a number of buffers depending on whether it is single-buffered, double-buffered, stereo, or stereo double-buffered. The components of a color buffer are therefore referred to as the front and back buffers. For devices that support stereo buffers, the color buffer is made up of the front left buffer, front right buffer, back left buffer, and back right buffer.

Color Map
A hardware implementation of a color look-up table. The frame buffer hardware uses each 8-bit pixel value as an index into this table when refreshing the physical display.

Color Model
The set of rules for manipulating color values in a processing system. IVL can be described as using an RGBA color model since it is based on processing pixel values with red, green, blue, and alpha components.

Color Table
An array that is used in a color look-up operation. Each incoming pixel component value is used as an index into this array. The value stored in the array at the indexed location is extracted and becomes the pixel component for subsequent processing.

Convolution
A common image-processing operation that can be used to filter an image. The filtering is accomplished by computing the sum of products between the source image and a smaller image or matrix called the **convolution filter** or **convolution kernel**. The convolution filter can be loaded with different values to achieve effects like sharpening, blurring, and edge detection.

Convolution Border Mode
An attribute that defines how IVL treats image borders during the convolution process. The border mode you choose may cause the output image to be a different size than the source image.

Convolution Filter
A two-dimensional image that is used during convolution to achieve effects like sharpening, blurring, and edge detection.

Convolution Kernel
Another name for convolution filter.

Cubic Weight Factor
A coefficient that provides additional control over the bicubic interpolation operation. The weighting factor biases the cubic curve used to perform the interpolation. Cubic weighting factors typically assume values from -1.0 to 0.0.

Current Drawable
The X window or GLX pixmap that contains the buffers that will be used for all subsequent rendering operations.

Current Draw Buffer
The buffer that is the target of all subsequent pixel-write operations. This term is sometimes shortened to "draw buffer".

Current Raster Position
The location at which to draw a pixel rectangle. This value is in window coordinates. This term is sometimes shortened to "raster position".

Current Read Buffer
The buffer from which pixels will be obtained during all subsequent pixel-read operations. This term is sometimes shortened to "read buffer".

Direct Connection
A connection that bypasses the X network transport mechanism in order to access the rendering hardware directly.

Double-Buffering
The process of using two buffers to provide smooth animation. This is done by drawing into the back buffer while the front buffer is displayed. Once drawing is complete, the contents of the back buffer are moved to the front buffer. See also **front buffer** and **back buffer**; compare with **single-buffering**.

Draw Buffer
The buffer that is the target of all subsequent pixel-write operations.

Drawable
An X window or a GLX pixmap.

Extensions
Capabilities that are not yet part of the OpenGL standard. The "`EXT`" suffix indicates OpenGL extensions that are supported by two or more vendors. The "`HP`" suffix indicates OpenGL extensions that are currently supported only by Hewlett-Packard.

Filter
Another name for **Convolution Filter**.

Fragment
A data structure containing pixel information which is a product of the pixel rasterization operation. A fragment consists of a color value and the coordinate of the frame buffer location at which that color value is to be written.

Fragment Operations
Operations that are applied to a fragment prior to writing the fragment's color value into the frame buffer. The two fragment operations that are currently defined for IVL are the **pixel ownership test** and the **scissor test**.

Frame Buffer
A two-dimensional array of memory locations that stores pixel values. Some of these locations correspond to the pixels that are visible on the display screen. Other locations are used for non-displayed pixel values, such as the back buffer of a double-buffered window.

Front Buffer
The displayed buffer in a double-buffer pair. See **double-buffering**; compare with **back buffer**.

GLX
The OpenGL extension to X. An X server extension that allows IVL to coordinate its rendering operations with those of X and other extensions.

GLX Pixmap
An X pixmap that has been enabled to support rendering via IVL. GLX pixmaps have an associated visual type, so they can contain the extended visual attributes defined by IVL.

Image
A rectangular array of pixel values, either in client memory or in the frame buffer.

Image Format
A term used to describe the organization of the pixel components in an image. IVL currently supports two formats: `GL_LUMINANCE`, which indicates that

pixel values are stored as luminance (one-component) values, and `GL_RGBA`, which indicates that pixel values are stored as red, green, blue, and alpha (four-component) values.

Image Transform
The stage of the IVL pixel processing pipeline that provides support for image scaling (zoom), rotating, translation (pan), and interpolation.

Image Type
The storage unit for each component of a pixel. Pixel components may be either `GL_UNSIGNED_BYTE` or `GL_UNSIGNED_SHORT`.

Interpolation
Another name for **resampling**.

IVL
Image Visualization Library. A 2D API for image processing with a programming interface very similar to the imaging portions of the OpenGL API together with several defined imaging extensions.

IVX
Image Visualization Accelerator. Optional graphics hardware that accelerates many of IVL's capabilities.

Logical Buffer
Bitplanes are grouped into logical buffers. The only logical buffer that is supported in this release of IVL is the color buffer.

Look-Up Table
See **color table**.

Luminance
A term used to describe images that contain only one pixel component per pixel. A one-component image is sometimes called a "grayscale" or "intensity" image.

Luminance Format
A one-component pixel format that contains only luminance pixel component values.

Machine Data Type
The data types that are "natural" for a computer architecture. The machine data types for a workstation may include signed and unsigned 8-bit values, signed and unsigned 16-bit values, signed and unsigned 32-bit values, 32-bit floating point values, and 64-bit floating point values.

Manhattan Distance
The distance between two points on a grid addressed by integer coordinates. This distance is the sum of the horizontal distance plus the vertical distance between the two points.

Monoscopic Window
A window that is not stereoscopic, i.e., does not support stereo viewing.

OpenGL
A 3D graphics API that includes capabilities for 2D imaging.

OpenGL Imaging Extensions
Extensions to OpenGL that support common image processing and display operations. See also **Extensions**.

Overlay Planes
A set of bitplanes that lie on top of the bitplanes for a color buffer. Applications can render into the overlay planes without disturbing the contents of the color buffer. The resulting display shows the contents of the overlay planes superimposed on the contents of the color buffer. Not all frame buffers include overlay planes.

Pixel Component
The fundamental element of a pixel. An RGBA pixel has four components: red, green, blue, and alpha. A luminance pixel has just one component (luminance).

Pixel Ownership Test
The test performed by IVL to determine if the pixel location at which the fragment is to be written is part of the current drawable. If it is not, the fragment is discarded.

Pixel Rasterization
The process by which the pixels of an image are converted to fragments, each corresponding to a pixel in the frame buffer.

Pixel Rectangle
A rectangular array of pixels, either in client memory or in the frame buffer. Used synonymously with "**image**".

Pixel Transfer
A set of operations that are applied whenever pixels are transferred from one place to another in the IVL environment. The operations include color table lookup, convolution, and image transformation.

Pixel Unpacking
The process of reading pixel values from host memory using the *type* and *format* parameters and the state values defined by `glPixelStore`.

Pixmap
A non-displayable region of the frame buffer into which rendering may occur. In the X Window System, a pixmap is defined to be very similar to a window. The X notion of a pixmap has to be extended in order to use pixmaps for IVL rendering. See **GLX pixmap**.

Post-Image Transform Color Table
The color look-up table that immediately follows the image transformation stage of the pixel processing pipeline.

Progressive Refinement
Rendering an image using multiple passes, with improved interpolation quality in each pass.

Raster Position
The location at which to draw a pixel rectangle. This value is in window coordinates.

Read Buffer
The buffer from which pixels will be obtained during subsequent pixel-read operations.

Reconstruction
The process that maps the discrete image samples computed by **resampling** into a continuous surface. In this process, pixel values between the sample points are computed using a method such as bilinear or bicubic interpolation. Compare with **Resampling**.

Rendering Context
The data structure that encapsulates server state information. The majority of IVL state is stored in the rendering context.

Resampling
The process of transforming a sampled image from one coordinate system to another. The two coordinate systems are related by the mapping function of the transformation. Using the inverse of the transformation, the regular grid corresponding to the pixel locations in the output image is mapped onto the input image. The input image is then sampled at each of these points, and the sampled values are assigned to their respective output locations in the output image. Compare with **Reconstruction**.

RGBA
A term used to describe images that contain four pixel components per pixel (red, green, blue, and alpha).

RGBA Format
A four-component pixel format that contains red, green, blue, and alpha pixel component values.

Scissor Box
A rectangular clipping region, defined in window coordinates. When the scissor test is enabled, only pixel locations within the scissor box can be modified.

Scissor Test
A clipping operation that eliminates any pixels that would be drawn outside of the current **scissor box**.

Server State
State that resides in the server and controls the rendering process. The current rendering context and the current drawable are among the few state values that are not part of server state.

Single-Buffering
A mode in which the same buffer is used simultaneously for drawing and displaying an image. In this mode, animations may reveal slight halting between frames. Compare with **double-buffering**.

Stereoscopic Window
A window that contains both a left eye view and a right eye view to support stereo viewing. When the proper viewing equipment is used, the user will see a three-dimensional, stereo image.

Subimage
A rectangular portion of a larger image.

Visual Type
The **X Visual** that defines the display attributes of a window. There are six visual types in X: `TrueColor`, `DirectColor`, `PseudoColor`, `StaticColor`, `GrayScale`, and `StaticGray`.

Window-Level Mapping
An image processing term that defines a continuous transfer function specifying the input to output intensity-value mapping. The window defines the width or range of input intensity values that will be mapped from black to white. The level defines where the center of the window range will fall with respect to the input intensity value domain. In IVL, support for window-level mapping is implemented as a color look-up table.

X Visual
The mechanism by which frame buffer features are exposed and software portability is defined in the X Window System. The `XVisualInfo` structure in X defines the visual type, screen, and depth of the window, as well as the number of bits of red, green, and blue information in a pixel.

Index

A

Abstract Machine, 2-10
Angle of Rotation, 5-10
API, Glossary-1
Architecture
  Hardware and Software, 2-9
Attributes, 3-6
Audience and Scope, 0-2

B

Back Buffer, Glossary-1
Bitplane, Glossary-1
Buffer
  Back, Glossary-1
  Color, Glossary-1
  Draw, Glossary-3
  Frame, Glossary-4
  Front, Glossary-4
  Logical, Glossary-5
  Read, Glossary-7

C

CDE and VUE, 1-3
Client State, Glossary-1
Clipping, Glossary-1
Clip Rectangles, 5-11
Color Buffer, Glossary-1
Color Map, Glossary-2
Color Map Management, 5-4, 5-7
Color Model, 2-4, Glossary-2
Color Table, 2-22, Glossary-2
Command
  `graphinfo`, 1-6
  `uname`, 1-4
  `what`, 1-4
Compiling, 3-8
Configuration, 1-2
`GL{\under }UNSIGNED{\under }BYTE` constant, B-25
`GL{\under }UNSIGNED{\under }SHORT` constant, B-25
Constants
  `GL{\under }UNSIGNED{\under }BYTE`, B-25
  `GL{\under }UNSIGNED{\under }SHORT`, B-25
  `GL_ALPHA_BITS`, B-32
  `GL_BACK`, B-21, B-22, B-68
  `GL_BACK_LEFT`, B-21, B-22, B-68
  `GL_BLUE_BITS`, B-32
  `GL_COLOR`, B-18, B-19
  `GL_COLOR_BUFFER_BIT`, B-2
  `GL_COLOR_CLEAR_VALUE`, B-3, B-4, B-32
  `GL_COLOR_TABLE_ALPHA_SIZE_EXT`, B-38, B-39
  `GL_COLOR_TABLE_BIAS_EXT`, B-6, B-8, B-38, B-39
  `GL_COLOR_TABLE_BLUE_SIZE_EXT`, B-38, B-39
  `GL_COLOR_TABLE_FORMAT_EXT`, B-38, B-39
  `GL_COLOR_TABLE_GREEN_SIZE_EXT`, B-38, B-39

GL_COLOR_TABLE_LUMINANCE_SIZE_EXT, B-38, B-39
GL_COLOR_TABLE_RED_SIZE_EXT, B-38, B-39
GL_COLOR_TABLE_SCALE_EXT, B-6, B-8, B-38, B-39
GL_COLOR_TABLE_WIDTH_EXT, B-38, B-39
GL_CONSTANT_BORDER_HP, B-13, B-14, B-15, B-16, B-17
GL_CONVOLUTION_2D_EXT, B-10, B-11, B-12, B-13, B-14, B-19, B-24, B-27, B-32, B-41, B-43, B-57
GL_CONVOLUTION_BORDER_COLOR_HP, B-14, B-15, B-16, B-17, B-43, B-44
GL_CONVOLUTION_BORDER_MODE_EXT, B-13, B-14, B-15, B-17, B-43, B-44
GL_CONVOLUTION_FILTER_BIAS_EXT, B-10, B-14, B-15, B-17, B-43, B-44
GL_CONVOLUTION_FILTER_SCALE_EXT, B-10, B-14, B-17, B-43, B-44
GL_CONVOLUTION_FORMAT_EXT, B-43, B-44
GL_CONVOLUTION_HEIGHT_EXT, B-43, B-44
GL_CONVOLUTION_WIDTH_EXT, B-43, B-44
GL_CUBIC_HP, B-52, B-53, B-54, B-55, B-56
GL_CURRENT_RASTER_POSITION, B-20, B-23, B-25, B-32, B-67
GL_CURRENT_RASTER_POSITION_VALID, B-20, B-25, B-32, B-66, B-67
GL_DOUBLEBUFFER, B-32
GL_DRAW_BUFFER, B-22, B-32
GL_EXTENSIONS, B-50
GL_FALSE, B-31, B-57
GL_FLOAT, B-10, B-41
GL_FRONT, B-21, B-22, B-68
GL_FRONT_LEFT, B-21, B-22, B-68
GL_GREEN_BITS, B-32
GL_IGNORE_BORDER_HP, B-13, B-14, B-15, B-17
GL_IMAGE_CUBIC_WEIGHT_HP, B-48, B-49, B-52, B-53, B-54
GL_IMAGE_MAG_FILTER_HP, B-48, B-49, B-52, B-53, B-54, B-56
GL_IMAGE_MIN_FILTER_HP, B-48, B-49, B-52, B-53, B-54, B-56
GL_IMAGE_ROTATE_ANGLE_HP, B-48, B-49, B-52, B-53, B-54
GL_IMAGE_ROTATE_ORIGIN_X_HP, B-48, B-49, B-52, B-53, B-54
GL_IMAGE_ROTATE_ORIGIN_Y_HP, B-48, B-49, B-52, B-53, B-54
GL_IMAGE_SCALE_X_HP, B-48, B-49, B-52, B-53, B-54
GL_IMAGE_SCALE_Y_HP, B-48, B-49, B-52, B-53, B-54
GL_IMAGE_TRANSFORM_2D_HP, B-27, B-35, B-48, B-49, B-52, B-53, B-55, B-57
GL_IMAGE_TRANSLATE_X_HP, B-48, B-49, B-52, B-53, B-54
GL_IMAGE_TRANSLATE_Y_HP, B-48, B-49, B-52, B-53, B-54
GL_INVALID_ENUM, B-7, B-9, B-11, B-12, B-17, B-20, B-22, B-25, B-28, B-35, B-37, B-40, B-42, B-45, B-46, B-49, B-51, B-56, B-58, B-62, B-65, B-69, B-72
GL_INVALID_OPERATION, B-22, B-46, B-69
GL_INVALID_VALUE, B-2, B-7, B-11, B-12, B-20, B-25, B-46, B-62, B-72, B-74
GL_LINEAR, B-52, B-53, B-55, B-56

GL_LUMINANCE, B-5, B-6, B-10, B-11, B-23, B-25, B-36, B-41, B-70, B-71
GL_LUMINANCE12_EXT, B-5
GL_LUMINANCE16_EXT, B-5
GL_LUMINANCE4_EXT, B-5, B-6
GL_LUMINANCE8_EXT, B-5
GL_MAX_CONVOLUTION_HEIGHT_EXT, B-12, B-43, B-44
GL_MAX_CONVOLUTION_WIDTH_EXT, B-11, B-43, B-44
GL_NEAREST, B-52, B-53, B-55, B-56
GL_NO_ERROR, B-46
GL_OUT_OF_MEMORY, B-47
GL_PACK_ALIGNMENT, B-33, B-59, B-60, B-61, B-62
GL_PACK_ROW_LENGTH, B-33, B-59, B-60, B-61, B-62, B-71
GL_PACK_SKIP_PIXELS, B-33, B-59, B-60, B-61, B-62
GL_PACK_SKIP_ROWS, B-33, B-59, B-60, B-61, B-62, B-71
GL_POST_CONVOLUTION_ALPHA_BIAS_EXT, B-33, B-63, B-64, B-65
GL_POST_CONVOLUTION_ALPHA_SCALE_EXT, B-33, B-63, B-64, B-65
GL_POST_CONVOLUTION_BLUE_BIAS_EXT, B-33, B-63, B-64, B-65
GL_POST_CONVOLUTION_BLUE_SCALE_EXT, B-33, B-63, B-64, B-65
GL_POST_CONVOLUTION_*c*, B-11
GL_POST_CONVOLUTION_GREEN_BIAS_EXT, B-33, B-63, B-64, B-65
GL_POST_CONVOLUTION_GREEN_SCALE_EXT, B-33, B-63, B-64, B-65
GL_POST_CONVOLUTION_RED_BIAS_EXT, B-33, B-63, B-64, B-65
GL_POST_CONVOLUTION_RED_SCALE_EXT, B-34, B-63, B-64, B-65
GL_POST_IMAGE_TRANSFORM_COLOR_TABLE_HP, B-5, B-6, B-7, B-8, B-9, B-27, B-34, B-36, B-38, B-39, B-57
GL_READ_BUFFER, B-34, B-69
GL_RED_BITS, B-34
GL_REDUCE_EXT, B-13, B-14, B-15, B-17
GL_RENDERER, B-50
GL_REPLICATE_BORDER_HP, B-13, B-14, B-16, B-17
GL_RGB, B-61
GL_RGB10_A2_EXT, B-5
GL_RGB5_A1_EXT, B-5
GL_RGBA, B-5, B-6, B-7, B-10, B-11, B-23, B-24, B-25, B-36, B-37, B-39, B-41, B-60, B-70, B-71, B-72
GL_RGBA12_EXT, B-5
GL_RGBA16_EXT, B-5
GL_RGBA2_EXT, B-5
GL_RGBA4_EXT, B-5
GL_RGBA8_EXT, B-5, B-6
GL_RGBA_MODE, B-34
GL_SCISSOR_BOX, B-34, B-74
GL_SCISSOR_TEST, B-27, B-34, B-57, B-73, B-74
GL_TABLE_TOO_LARGE_EXT, B-7, B-47
GL_TRUE, B-31, B-57
GL_UNPACK_ALIGNMENT, B-23, B-34, B-59, B-61, B-62
GL_UNPACK_ROW_LENGTH, B-34, B-59, B-60, B-61, B-62
GL_UNPACK_SKIP_PIXELS, B-34, B-59, B-61, B-62

GL_UNPACK_SKIP_ROWS, B-34, B-59, B-61, B-62
GL_UNSIGNED_BYTE, B-5, B-23, B-36, B-70, B-71
GL_UNSIGNED_SHORT, B-5, B-7, B-23, B-25, B-36, B-37, B-70, B-71, B-72
GL_VENDOR, B-50
GL_VERSION, B-50
GL_WRAP_BORDER_HP, B-13, B-14, B-16, B-17
GLX_ALPHA_SIZE, B-80, B-88, B-89
GLX_BAD_ATTRIB, B-90
GLX_BAD_CONTEXT, B-86, B-92
GLX_BAD_CURRENT_WINDOW, B-92, B-95
GLX_BAD_DRAWABLE, B-92, B-95
GLX_BAD_PIXMAP, B-87
GLX_BAD_SCREEN, B-90
GLX_BAD_VISUAL, B-90
GLX_BLUE_SIZE, B-80, B-88, B-89
GLX_BUFFER_SIZE, B-81, B-85, B-88, B-89
GLX_DOUBLEBUFFER, B-79, B-89
GLX_GREEN_SIZE, B-80, B-88, B-89
GLX_LEVEL, B-79, B-88
GLX_NO_EXTENSION, B-90
GLX_RED_SIZE, B-80, B-88, B-89
GLX_RGBA, B-79, B-88, B-89
GLX_USE_GL, B-88, B-90
GLX_X_VISUAL_TYPE_EXT, B-80, B-89
Contents of Manual, 0-1
Conventions
  Naming, 3-1
Conversion to Frame Buffer Resolution, 2-22
Convolution, 2-18, Glossary-2
Convolution Border Mode, Glossary-2
Convolution Filter, Glossary-2
Convolution Kernel, Glossary-2
Cubic Weight Factor, Glossary-2
Current Drawable, Glossary-3
Current Draw Buffer, Glossary-3
Current Raster Position, Glossary-3
Current Read Buffer, Glossary-3

D

Device
  Description, 5-10
  Descriptions, 5-6
Device Description, 5-3
Devices, 5-1
Direct Connection, Glossary-3
Distributed Environments, 3-10
Double-Buffering, 1-2, Glossary-3
Double-Buffering Support, 4-7
Drawable, Glossary-3
  Current, Glossary-3
Draw Buffer, Glossary-3

E

Entry-Level Color Graphics Devices, 5-3
Error Handling, 3-15
Extensions, Glossary-3
  Supported by HP Only, 3-2
  Supported by Multiple Vendors, 3-2
Extensions to IVL, 3-2

F

Filesets, 1-1
Filter
  Convolution, Glossary-2
Format
  Image, Glossary-4
  Luminance, Glossary-5
Formatting Conventions, 0-2
For More Information, 0-3
Fragment, Glossary-4
  Operations, 2-16
Fragment Operations, 2-23, Glossary-4
Frame Buffer, Glossary-4

Organization, 2-5
Resolution, 2-22
Writing, 2-17
Front Buffer, Glossary-4

G

General Performance Hints, 3-13
GL_ALPHA_BITS constant, B-32
GL_BACK constant, B-21, B-22, B-68
GL_BACK_LEFT constant, B-21, B-22, B-68
GL_BLUE_BITS constant, B-32
GL_COLOR_BUFFER_BIT constant, B-2
GL_COLOR_CLEAR_VALUE constant, B-3, B-4, B-32
GL_COLOR constant, B-18, B-19
GL_COLOR_TABLE_ALPHA_SIZE_EXT constant, B-38, B-39
GL_COLOR_TABLE_BIAS_EXT constant, B-6, B-8, B-38, B-39
GL_COLOR_TABLE_BLUE_SIZE_EXT constant, B-38, B-39
GL_COLOR_TABLE_FORMAT_EXT constant, B-38, B-39
GL_COLOR_TABLE_GREEN_SIZE_EXT constant, B-38, B-39
GL_COLOR_TABLE_LUMINANCE _SIZE_EXT constant, B-38, B-39
GL_COLOR_TABLE_RED_SIZE_EXT constant, B-38, B-39
GL_COLOR_TABLE_SCALE_EXT constant, B-6, B-8, B-38, B-39
GL_COLOR_TABLE_WIDTH_EXT constant, B-38, B-39
GL_CONSTANT_BORDER_HP constant, B-13, B-14, B-15, B-16, B-17
GL_CONVOLUTION_2D_EXT constant, B-10, B-11, B-12, B-13, B-14, B-19, B-24, B-27, B-32, B-41, B-43, B-57
GL_CONVOLUTION_BORDER_COLOR_HP constant, B-14, B-15, B-16, B-17, B-43, B-44
GL_CONVOLUTION_BORDER_MODE_EXT constant, B-13, B-14, B-15, B-17, B-43, B-44
GL_CONVOLUTION_FILTER_BIAS_EXT constant, B-10, B-14, B-15, B-17, B-43, B-44
GL_CONVOLUTION_FILTER_SCALE_EXT constant, B-10, B-14, B-17, B-43, B-44
GL_CONVOLUTION_FORMAT_EXT constant, B-43, B-44
GL_CONVOLUTION_HEIGHT_EXT constant, B-43, B-44
GL_CONVOLUTION_WIDTH_EXT constant, B-43, B-44
GL_CUBIC_HP constant, B-52, B-53, B-54, B-55, B-56
GL_CURRENT_RASTER_POSITION constant, B-20, B-23, B-25, B-32, B-67
GL_CURRENT_RASTER_ POSITION_VALID constant, B-20, B-25, B-32, B-66, B-67
GL_DOUBLEBUFFER constant, B-32
GL_DRAW_BUFFER constant, B-22, B-32
GL_EXTENSIONS constant, B-50
GL_FALSE constant, B-31, B-57
glFinish Routine, 4-8
GL_FLOAT constant, B-10, B-41
glFlush Routine, 4-9
GL_FRONT constant, B-21, B-22, B-68
GL_FRONT_LEFT constant, B-21, B-22, B-68
GL_GREEN_BITS constant, B-32
GL_IGNORE_BORDER_HP constant, B-13, B-14, B-15, B-17
GL_IMAGE_CUBIC_WEIGHT_HP constant, B-48, B-49, B-52, B-53, B-54

GL_IMAGE_MAG_FILTER_HP constant, B-48, B-49, B-52, B-53, B-54, B-56
GL_IMAGE_MIN_FILTER_HP constant, B-48, B-49, B-52, B-53, B-54, B-56
GL_IMAGE_ROTATE_ANGLE_HP constant, B-48, B-49, B-52, B-53, B-54
GL_IMAGE_ROTATE_ORIGIN_X_HP constant, B-48, B-49, B-52, B-53, B-54
GL_IMAGE_ROTATE_ORIGIN_Y_HP constant, B-48, B-49, B-52, B-53, B-54
GL_IMAGE_SCALE_X_HP constant, B-48, B-49, B-52, B-53, B-54
GL_IMAGE_SCALE_Y_HP constant, B-48, B-49, B-52, B-53, B-54
GL_IMAGE_TRANSFORM_2D_HP constant, B-27, B-35, B-48, B-49, B-52, B-53, B-55, B-57
GL_IMAGE_TRANSLATE_X_HP constant, B-48, B-49, B-52, B-53, B-54
GL_IMAGE_TRANSLATE_Y_HP constant, B-48, B-49, B-52, B-53, B-54
GL_INVALID_ENUM constant, B-7, B-9, B-11, B-12, B-17, B-20, B-22, B-25, B-28, B-35, B-37, B-40, B-42, B-45, B-46, B-49, B-51, B-56, B-58, B-62, B-65, B-69, B-72
GL_INVALID_OPERATION constant, B-22, B-46, B-69
GL_INVALID_VALUE constant, B-2, B-7, B-11, B-12, B-20, B-25, B-46, B-62, B-72, B-74
GL_LINEAR constant, B-52, B-53, B-55, B-56
GL_LUMINANCE12_EXT constant, B-5
GL_LUMINANCE16_EXT constant, B-5
GL_LUMINANCE4_EXT constant, B-5, B-6
GL_LUMINANCE8_EXT constant, B-5
GL_LUMINANCE constant, B-5, B-6, B-10, B-11, B-23, B-25, B-36, B-41, B-70, B-71
GL_MAX_CONVOLUTION_HEIGHT_EXT constant, B-12, B-43, B-44
GL_MAX_CONVOLUTION_WIDTH_EXT constant, B-11, B-43, B-44
GL_NEAREST constant, B-52, B-53, B-55, B-56
GL_NO_ERROR constant, B-46
GL_OUT_OF_MEMORY constant, B-47
GL_PACK_ALIGNMENT constant, B-33, B-59, B-60, B-61, B-62
GL_PACK_ROW_LENGTH constant, B-33, B-59, B-60, B-61, B-62, B-71
GL_PACK_SKIP_PIXELS constant, B-33, B-59, B-60, B-61, B-62
GL_PACK_SKIP_ROWS constant, B-33, B-59, B-60, B-61, B-62, B-71
GL_POST_CONVOLUTION_ ALPHA_BIAS_EXT constant, B-33, B-63, B-64, B-65
GL_POST_CONVOLUTION_ ALPHA_SCALE_EXT constant, B-33, B-63, B-64, B-65
GL_POST_CONVOLUTION_ BLUE_BIAS_EXT constant, B-33, B-63, B-64, B-65
GL_POST_CONVOLUTION_ BLUE_SCALE_EXT constant, B-33, B-63, B-64, B-65
GL_POST_CONVOLUTION_*c* constant, B-11
GL_POST_CONVOLUTION_ GREEN_BIAS_EXT constant, B-33, B-63, B-64, B-65
GL_POST_CONVOLUTION_ GREEN_SCALE_EXT constant, B-33, B-63, B-64, B-65
GL_POST_CONVOLUTION_ RED_BIAS_EXT constant, B-33, B-63, B-64, B-65
GL_POST_CONVOLUTION_ RED_SCALE_EXT constant, B-34, B-63, B-64, B-65

GL_POST_IMAGE_TRANSFORM_
COLOR_TABLE_HP constant, B-5, B-6, B-7, B-8, B-9, B-27, B-34, B-36, B-38, B-39, B-57
GL_READ_BUFFER constant, B-34, B-69
GL_RED_BITS constant, B-34
GL_REDUCE_EXT constant, B-13, B-14, B-15, B-17
GL_RENDERER constant, B-50
GL_REPLICATE_BORDER_HP constant, B-13, B-14, B-16, B-17
GL_RGB10_A2_EXT constant, B-5
GL_RGB5_A1_EXT constant, B-5
GL_RGBA12_EXT constant, B-5
GL_RGBA16_EXT constant, B-5
GL_RGBA2_EXT constant, B-5
GL_RGBA4_EXT constant, B-5
GL_RGBA8_EXT constant, B-5, B-6
GL_RGBA constant, B-5, B-6, B-7, B-10, B-11, B-23, B-24, B-25, B-36, B-37, B-39, B-41, B-60, B-70, B-71, B-72
GL_RGBA_MODE constant, B-34
GL_RGB constant, B-61
GL_SCISSOR_BOX constant, B-34, B-74
GL_SCISSOR_TEST constant, B-27, B-34, B-57, B-73, B-74
GL_TABLE_TOO_LARGE_EXT constant, B-7, B-47
GL_TRUE constant, B-31, B-57
GL_UNPACK_ALIGNMENT constant, B-23, B-34, B-59, B-61, B-62
GL_UNPACK_ROW_LENGTH constant, B-34, B-59, B-60, B-61, B-62
GL_UNPACK_SKIP_PIXELS constant, B-34, B-59, B-61, B-62
GL_UNPACK_SKIP_ROWS constant, B-34, B-59, B-61, B-62
GL_UNSIGNED_BYTE constant, B-5, B-23, B-36, B-70, B-71
GL_UNSIGNED_SHORT constant, B-5, B-7, B-23, B-25, B-36, B-37, B-70, B-71, B-72
GL_VENDOR constant, B-50
GL_VERSION constant, B-50
GL_WRAP_BORDER_HP constant, B-13, B-14, B-16, B-17
GLX, Glossary-4
'GLX_ALPHA_SIZE'' constant, B-80, B-88, B-89
'GLX_BAD_ATTRIB'' constant, B-90
'GLX_BAD_CONTEXT'' constant, B-86, B-92
'GLX_BAD_CURRENT_WINDOW'' constant, B-92, B-95
'GLX_BAD_DRAWABLE'' constant, B-92, B-95
'GLX_BAD_PIXMAP'' constant, B-87
'GLX_BAD_SCREEN'' constant, B-90
'GLX_BAD_VISUAL'' constant, B-90
'GLX_BLUE_SIZE'' constant, B-80, B-88, B-89
'GLX_BUFFER_SIZE'' constant, B-81, B-85, B-88, B-89
'GLX_DOUBLEBUFFER'' constant, B-79, B-89
'GLX_GREEN_SIZE'' constant, B-80, B-88, B-89
'GLX_LEVEL'' constant, B-79, B-88
'GLX_NO_EXTENSION'' constant, B-90
GLX Pixmap, Glossary-4
'GLX_RED_SIZE'' constant, B-80, B-88, B-89
'GLX_RGBA'' constant, B-79, B-88, B-89
'GLX_USE_GL'' constant, B-88, B-90
'GLX_X_VISUAL_TYPE_EXT'' constant, B-80, B-89
graphinfo Command, 1-6

H

Hardware Architecture, 2-9
HCRX-24, 5-7
HCRX-24 Description, 5-6
HCRX-8, 5-7
HCRX-8 Description, 5-6
HCRX Family Device Descriptions, 5-5
High-Level IVL Overview, 2-7
HP Image Library, 2-3
HP-PHIGS, 2-3

I

If You Have Incompatible Software, 1-4
Image, Glossary-4
  Data Formatting, 3-12
  Transform, 2-21
Image Format, Glossary-4
Image Library, 2-3
Image Transform, Glossary-5
Image Type, Glossary-5
Image Visualization Accelerator Device Description, 5-10
Imaging Operations, 3-6
Implementation Restrictions, 3-10
Incompatible Software, 1-4
Information on Revision, 1-4
Installation, 1-1
Interpolation, Glossary-5
IVL, Glossary-5
  And Backing Store, 4-8
  and Color Recovery, 4-6
  API Routines, 3-6
  Constants, 3-1, 3-5
  Data Type Names, 3-4
  Data Types, 3-3
  Description, 2-1
  Extensions, 3-2
  Implementations, 5-1
  Machine, 2-10
  Overview, 2-7
  Relationship to OpenGL, 2-2
  Routines, 3-1, 3-4
  What is it?, 2-1
  With other graphics APIs, 2-2
IVL Data , 3-8
IVL Filesets, 1-1
IVX, Glossary-5

K

Kernel
  Convolution, Glossary-2

L

Library
  Image, 2-3
Linking, 3-8
Logical Buffer, Glossary-5
Look-Up Table, Glossary-5
Luminance, Glossary-5
Luminance Format, Glossary-5

M

Machine
  Abstract, 2-10
Machine Data Type, Glossary-6
Managing Rendering Contexts, 4-6
Manhattan Distance, Glossary-6
Manual Contents, 0-1
Model
  Color, Glossary-2
Monoscopic Window, Glossary-6
Motif, 1-3
Multi-Threaded Applications, 3-10

N

Naming Conventions, 3-1

O

OpenGL, Glossary-6
OpenGL Imaging Extensions, Glossary-6
OpenGL Implementations, 2-3
Overlay Planes, Glossary-6

Overlay Transparency, 5-4, 5-8
Overlay Transparency with HCRX-24 Devices, 5-9
Overlay Transparency with HCRX-8 Devices, 5-8

P

Performance Hints, 5-11
Performance Hints for Workstations with IVX Hardware, 3-13
Performance Hints for Workstations without IVX Hardware, 3-14
Performance Tuning Tips, 3-13
PEX, 2-3
Pipeline Stages, 2-12
Pixel
  Component, Glossary-6
  Ownership Test, Glossary-6
  Rasterization, 2-16, Glossary-7
  Rectangle, Glossary-7
  Transfer, 2-16, 2-17, Glossary-7
  Unpack, 2-12
  Unpacking, 3-9, Glossary-7
Pixel Ownership Test, 2-23
Pixel Rasterization, 2-23
Pixmap, Glossary-7
Pixmaps, 4-8
Planes
  Overlay, Glossary-6
Post-Convolution Scale and Bias, 2-21
Post-Image Transform Color Table, 2-22, Glossary-7
Printing the IVL Documentation, 0-4
Programming Advice, 3-11
Progressive Refinement, Glossary-7

Q

Query Data Values, 3-11

R

Rasterization, 5-10
Raster Position, Glossary-7
Read Buffer, Glossary-7
Reconstruction, Glossary-8
Renderer
  Names, 5-2
Rendering Context, Glossary-8
Rendering Contexts, 2-8, 4-6
Resampling, Glossary-8
Revision Information, 1-4
RGBA, Glossary-8
RGBA Data Format, 3-12
RGBA Format, Glossary-8
Routine
  `glFinish`, 4-8
  `glFlush`, 4-9

S

Sample Code, 3-19
Scissor Box, Glossary-8
Scissor Test, 2-24, Glossary-8
Scope and Audience, 0-2
SD-UX, 1-1
Selecting Visuals, 4-5
Server State, Glossary-8
Setting and Querying Attributes, 3-6
Setting Up an X/IVL Program, 4-4
Single-Buffering, Glossary-9
Single Logical Screen, 1-2
Software, 1-4
Software Architecture, 2-9
Software versus Hardware-Accelerated Paths, 5-11
Standard IVL Routines and Constants, 3-1
Starbase, 2-3
State
  Client, Glossary-1
Stereoscopic Window, Glossary-9
Subimage, Glossary-9

Supported Data Formats, 3-8
Synchronization, 4-8
System
  Window Coordinate, 2-7

T

Table
  Color, Glossary-2
Transform
  Image, Glossary-5
Type
  Image, Glossary-5

U

`uname` Command, 1-4
Underlays, 3-10
Unpack Pixels, 2-12
Using Pixmaps, 4-8
Using SD-UX, 1-1
Using the `glFinish` Routine, 4-8
Using the `glFlush` Routine, 4-9
Using the `graphinfo` Command, 1-6
Using the `uname` Command, 1-4
Using the `what` Command, 1-4

V

Viewing IVL Documentation with Web Browsers, 0-4
Visual
  Default, 5-8
  Type, Glossary-9
  X, Glossary-9
Visuals, 4-5, 5-3, 5-7
VUE and CDE, 1-3

W

`what` Command, 1-4
What is IVL?, 2-1
Window Coordinate System, 2-7
Window-Level Mapping, Glossary-9
Window System Interaction, 3-7
Window System Routines and Constants, 3-2

X

X11, 1-2
X Configuration, 1-2
X Interaction, 4-1
Xlib and Motif, 2-2
X Visual, Glossary-9
X Windows Capabilities, 4-4